The following paragraph does not apply to the United Kingdom or any country or state where such provisions are inconsistent with local law.

The specifications in this manual are subject to change without notice. This manual is provided “AS IS”. International Business Machines Corp. makes no warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

International Business Machines Corp. does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

Address comments to IBM Corporation, 11400 Burnett Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM®
Power ISA
PowerPC®
Power Architecture
PowerPC Architecture
Power Family
RISC/System 6000®
POWER®
POWER2
POWER4
POWER4+
POWER5
POWER5+
POWER6®
POWER7®
System/370
System z

The POWER ARCHITECTURE and POWER.ORG. word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

AltiVec is a trademark of Freescale Semiconductor, Inc. used under license.

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.
Preface

The roots of the Power ISA (Instruction Set Architecture) extend back over a quarter of a century, to IBM Research. The POWER (Performance Optimization With Enhanced RISC) Architecture was introduced with the RISC System/6000 product family in early 1990. In 1991, Apple, IBM, and Motorola began the collaboration to evolve to the PowerPC Architecture, expanding the architecture's applicability. In 1997, Motorola and IBM began another collaboration, focused on optimizing PowerPC for embedded systems, which produced Book E.

In 2006, Freescale and IBM collaborated on the creation of the Power ISA Version 2.03, which represented the reunification of the architecture by combining Book E content with the more general purpose PowerPC Version 2.02. A significant benefit of the reunification is the establishment of a single, compatible, 64-bit programming model. The combining also extends explicit architectural endorsement and control to Auxiliary Processing Units (APUs), units of function that were originally developed as implementation- or product family-specific extensions in the context of the Book E allocated opcode space. With the resulting architectural superset comes a framework that clearly establishes requirements and identifies options.

To a very large extent, application program compatibility has been maintained throughout the history of the architecture, with the main exception being application exploitation of APUs. The framework identifies the base, pervasive, part of the architecture, and differentiates it from “categories” of optional function (see Section 1.3.5 of Book I). Because of the substantial differences in the supervisor (privileged) architecture that developed as Book E was optimized for embedded systems, the supervisor architectures for embedded and general purpose implementations are represented as mutually exclusive categories. Future versions of the architecture will seek to converge on a common solution where possible.

This document defines the Power ISA Version 2.07. It is comprised of five books and a set of appendices.

Book I, Power ISA User Instruction Set Architecture, covers the base instruction set and related facilities available to the application programmer. It includes five chapters derived from APU function, including the vector extension also known as Altivec.

Book II, Power ISA Virtual Environment Architecture, defines the storage model and related instructions and facilities available to the application programmer.

Book III-S, Power ISA Operating Environment Architecture, defines the supervisor instructions and related facilities used for general purpose implementations.

Book III-E, Power ISA Operating Environment Architecture, defines the supervisor instructions and related facilities used for embedded implementations. It was derived from Book E and extended to include APU function.

Book VLE, Power ISA Variable Length Encoded Instructions Architecture, defines alternative instruction encodings and definitions intended to increase instruction density for very low end implementations. It was derived from an APU description developed by Freescale Semiconductor.

As used in this document, the term “Power ISA” refers to the instructions and facilities described in Books I, II, III-S, III-E, and VLE.

Usage of the phrase “Book III” refers to both Book III-S and Book III-E. An exception to this rule is when, at the beginning of a Section or Book, it is specified that usage of the phrase “Book III” implies only either “Book III-S” or “Book III-E”.

Change bars have been included to indicate changes from the Power ISA Version 2.06B.
Summary of Changes in Power ISA Version 2.07

Version 2.07 of the PowerISA was created by applying the following requests for change (RFCs) to PowerISA Version 2.06B.

Performance Monitor Facility: Adds various performance monitoring facilities and a branch history buffer to Server architecture.

VSX Scalar Single-Precision: Adds support for scalar single-precision to VSX.

Transactional Memory: Adds support for a transactional memory storage model which allows an application to perform a sequence of accesses that appear to occur atomically with respect to other threads.

Processor Control Enhancements: Enables privileged and hypervisor software to send messages to other threads.

Instruction Cache Block Touch: The icbt instruction has been moved from the Embedded category to the Base category.

Extended Problem State Priority: Provides a mechanism that enables application programs to temporarily boost their priority.

Virtual Page Class Key Extensions for Instructions: Adds a new SPR similar to the AMR that controls whether instructions can be fetched from virtual addresses.

Reserved Bit Behavior Restriction and Processor Compatibility Register: Updates the PCR to accommodate new problem state features for PowerISA Version 2.07. In addition, requirements for reserved bit behaviors have been tightened in order to improve software compatibility across implementations.

Chip Information Register: Adds a privileged SPR which enables software to determine information about the chip on which the processor is implemented.

Crypto Operations: Adds instructions supporting AES, GCM, and SHA encryption and decryption, as well as a variety of CRCs and other finite field arithmetic operations.

VMX 64-Bit Integer Operations: Adds instructions supporting 2-way SIMD 64-bit integer operations.

Vector Miscellaneous Instructions: Adds SIMD count leading zeros instructions and a bit gather instruction.

Cache Hint Indicating Block Not Needed: Enables software to provide a cache hint indicating that it will no longer access a block.

Remove LPES1 from the LPCR: Eliminates the capability to request the processor to behave as previous versions of the architecture required when LPES1 = 0.

Split Endian Control: Adds a new MSR bit (SLE) as well as the mtisle instruction which allows its manipulation in problem state. This bit now allows the Endian values used for data and instruction storage accesses to be specified independently.

Direct Move Instructions: Adds new VSX instructions which remove the restriction of cross-register file moves.

Move dnh from Enhanced Debug to Embedded Category: The dnh instruction has been moved to the Embedded category.

Real Mode Storage Control for Instruction Fetching and Loosely-Related Caching-Inhibited Load/Store Changes: Defines an alternative to the existing instruction fetch RMSC approach whereby a first access to a region of storage will be performed as guarded, but subsequent accesses will be performed as non-guarded based on the success of the first.

Elemental Memory Barriers: Extends the definition of the existing sync instruction by adding several memory ordering functions.

Event-Based Branch Facility: A problem-state accessible event-based branch mechanism analogous to the interrupt mechanism is defined.

Stream Prefetch Changes: Allows DSCR access in problem state and adds additional stream prefetch functionality.

Add lqarx/stqcx. Instructions: Adds support for quad-word atomic storage operations.

Allow lq/stq in Problem State: Allows problem state software to use the lq and stq instructions.

Allow lq/stq in Little-Endian Mode: Removes the restriction that the lq and stq instructions can only operate in Big-Endian mode.

Add makeitso Instruction: Adds the miso extended mnemonic which allows producers in producer-consumer applications to provide a hint to push store data out.

Branch Conditional to Target Address Register: Defines a new instruction which branches conditionally to an address contained in a new SPR.

Remove DABR[X], add DAWR[X] and IABR SPRs: Removes and replaces existing DABR/DABRX SPRs with enhanced watchpoint functionality. Additionally adds instruction address breakpoint capability to the ISA.

Facility Availability Registers and Interrupts: A privileged register and a hypervisor register that enables various facilities are defined.

Reserved SPRs: Defines a set of reserved SPRs treated as no-ops in the current architecture so that
exploitation of new function in future designs can proceed more quickly and pervasively.

**Instruction Counter and Virtual Time Base**: Two registers, one that counts instructions completed by a thread, and another that counts at the same rate as the Time Base are added.

**Architecture Changes to Support Program Portability**: Eliminates software exposure to errors caused by variations in behavior due to implementation-dependent bits in CTRL and PPR.

**Miscellaneous Changes**: Various minor editorial corrections are made.

**Interrupts and Relocation, MMIO Emulation**: NewLPCR bits enable most interrupts to be taken with relocation on, and virtual page class key faults to cause Hypervisor Data Storage interrupts.

**Guest Timer Interrupts and Facilities**: Adds guest timer facilities to enable performance measurements for guest operating systems.

**VSX Unaligned Vector Storage Accesses**: Change VSX vector storage access instructions to support to byte-aligned addresses. Simplify Table 2, “Effect of Operand Placement on Performance,” on page 753 in Book II.

**VMX Miscellaneous Operations II**: Adds new instructions `vclzd`, `vpopcntb`, `vpopcnth`, `vpopcntw`, `vpopcntd`, `veqv`, `vnand`, and `vorc`.

**BFP/VSX Miscellaneous Operations**: Introduces new VSX instructions to address IEEE-754-2008 compliance when performing simple assignments between single-precision scalar and vector elements.

**VMX 32-bit Multiply Operations**: Introduces 4-way SIMD 32-bit integer multiply instructions.

**VMX Decimal Integer Operations**: Introduces new packed decimal add and subtract instructions.

**Remove Data Value Compare**: Since the data value compare function can be easily emulated within a data address compare handler, the data value compare registers are removed from Book III-E.

**VMX 128-bit Integer Operations**: Introduces new quad-word integer add and subtract instructions.

**Cache Lock Query Instructions**: Adds instructions to determine whether a cache block has been successfully locked with a preceding cache locking instruction.

**Embedded Guest Performance Monitor Interrupt**: Introduces a new Performance Monitor Interrupt that enables a Performance Monitor interrupt to be taken in guest state.
# Table of Contents

Preface ........................................ iii
  Summary of Changes in Power ISA Version 2.07 ..................... iv

Table of Contents ......................... vii

Figures ........................................ xxv

Book I: Power ISA User Instruction Set Architecture ............... 1

Chapter 1. Introduction ................. 3
  1.1 Overview.................................. 3
  1.2 Instruction Mnemonics and Operands 3
  1.3 Document Conventions ................. 3
  1.3.1 Definitions ............................. 3
  1.3.2 Notation ................................ 4
  1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs .......... 5
  1.3.4 Description of Instruction Operation 6
  1.3.5 Categories ............................... 8
  1.3.5.1 Phased-In/Phased-Out ................. 9
  1.3.5.2 Corequisite Category ................. 10
  1.3.5.3 Category Notation .................. 10
  1.3.6 Environments ............................ 10
  1.4 Processor Overview ..................... 11
  1.5 Computation modes ....................... 13
  1.5.1 Modes [Category: Server] ............ 13
  1.5.2 Modes [Category: Embedded] ......... 13
  1.6 Instruction Formats ..................... 13
  1.6.1 I-FORM .................................. 14
  1.6.2 B-FORM .................................. 14
  1.6.3 SC-FORM .................................. 14
  1.6.4 D-FORM .................................. 14
  1.6.5 DS-FORM .................................. 14
  1.6.6 DQ-FORM .................................. 14
  1.6.7 X-FORM .................................. 15
  1.6.8 XL-FORM .................................. 15
  1.6.9 XFX-FORM .................................. 15
  1.6.10 XFL-FORM ................................ 16
  1.6.11 XX1-FORM ................................ 16
  1.6.12 XX2-FORM ................................ 16
  1.6.13 XX3-FORM ................................ 16
  1.6.14 XX4-FORM ................................ 16
  1.6.15 XS-FORM ................................ 16
  1.6.16 XO-FORM ................................ 16
  1.6.17 A-FORM ................................ 16
  1.6.18 M-FORM ................................ 16
  1.6.19 MD-FORM ................................ 16
  1.6.20 MDS-FORM ................................ 16
  1.6.21 VA-FORM ................................ 16
  1.6.22 VC-FORM ................................ 16
  1.6.23 VX-FORM ................................ 17
  1.6.24 EVX-FORM ................................ 17
  1.6.25 EVS-FORM ................................ 17
  1.6.26 Z22-FORM ................................ 17
  1.6.27 Z23-FORM ................................ 17
  1.6.28 Instruction Fields .................... 17
  1.7 Classes of Instructions ............... 21
  1.7.1 Defined Instruction Class .......... 21
  1.7.2 Illegal Instruction Class .......... 21
  1.7.3 Reserved Instruction Class .......... 21
  1.8 Forms of Defined Instructions ....... 22
  1.8.1 Preferred Instruction Forms .......... 22
  1.8.2 Invalid Instruction Forms .......... 22
  1.8.3 Reserved-no-op Instructions [Category: Phased-In] .......... 22
  1.9 Exceptions ................................. 22
  1.10 Storage Addressing ..................... 23
  1.10.1 Storage Operands ..................... 23
  1.10.2 Instruction Fetches ................... 24
  1.10.3 Effective Address Calculation ....... 26

Chapter 2. Branch Facility ............... 29
  2.1 Branch Facility Overview .............. 29
  2.2 Instruction Execution Order ........... 29
  2.3 Branch Facility Registers ............. 30
  2.3.1 Condition Register .................... 30
  2.3.2 Link Register ........................... 32
  2.3.3 Count Register ............................ 32
  2.3.4 Target Address Register ............. 32
  2.4 Branch History Rolling Buffer [Category: Server] ............. 32
  2.4.1 Branch History Rolling Buffer Entry Format .......... 33
  2.5 Branch Instructions ..................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F.2.4</td>
<td>Conversion from Floating-Point Number to Signed Fixed-Point Integer Word</td>
<td>726</td>
</tr>
<tr>
<td>F.2.5</td>
<td>Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word</td>
<td>727</td>
</tr>
<tr>
<td>F.2.6</td>
<td>Conversion from Signed Fixed-Point Integer Doubleword to Floating-Point Number</td>
<td>727</td>
</tr>
<tr>
<td>F.2.7</td>
<td>Conversion from Unsigned Fixed-Point Integer Doubleword to Floating-Point Number</td>
<td>727</td>
</tr>
<tr>
<td>F.2.8</td>
<td>Conversion from Signed Fixed-Point Integer Word to Floating-Point Number</td>
<td>727</td>
</tr>
<tr>
<td>F.2.9</td>
<td>Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number</td>
<td>727</td>
</tr>
<tr>
<td>F.2.10</td>
<td>Unsigned Single-Precision BCD Arithmetic</td>
<td>728</td>
</tr>
<tr>
<td>F.2.11</td>
<td>Signed Single-Precision BCD Arithmetic</td>
<td>728</td>
</tr>
<tr>
<td>F.2.12</td>
<td>Unsigned Extended-Precision BCD Arithmetic</td>
<td>728</td>
</tr>
<tr>
<td>F.3</td>
<td>Floating-Point Selection [Category: Floating-Point]</td>
<td>730</td>
</tr>
<tr>
<td>F.3.1</td>
<td>Comparison to Zero</td>
<td>730</td>
</tr>
<tr>
<td>F.3.2</td>
<td>Minimum and Maximum</td>
<td>730</td>
</tr>
<tr>
<td>F.3.3</td>
<td>Simple if-then-else Constructions</td>
<td>730</td>
</tr>
<tr>
<td>F.3.4</td>
<td>Notes</td>
<td>730</td>
</tr>
<tr>
<td>F.4</td>
<td>Vector Unaligned Storage Operations [Category: Vector]</td>
<td>731</td>
</tr>
<tr>
<td>F.4.1</td>
<td>Loading a Unaligned Quadword Using Permute from Big-Endian Storage</td>
<td>731</td>
</tr>
</tbody>
</table>

**Book II:**

**Power ISA Virtual Environment Architecture**

**Chapter 1. Storage Model**

1.1 Definitions .................................. 735
1.2 Introduction .................................. 736
1.3 Virtual Storage ................................ 736
1.4 Single-Copy Atomicity .......................... 737
1.5 Cache Model ................................... 737
1.6 Storage Control Attributes ..................... 738
1.6.1 Write Through Required ....................... 738
1.6.2 Caching Inhibited ............................ 738
1.6.3 Memory Coherence Required [Category: Memory Coherence] .............. 739
1.6.4 Guarded ...................................... 739
1.6.5 Endianness [Category: Embedded.Little-Endian] ....................... 740

1.6.6 Variable Length Encoded (VLE) Instructions .................. 740
1.6.7 Strong Access Order [Category: SAO] ..................... 741
1.7 Shared Storage .................................. 742
1.7.1 Storage Access Ordering ........................ 742
1.7.2 Storage Ordering of I/O Accesses ..................... 744
1.7.3 Atomic Update .................................. 744
1.7.3.1 Reservations ................................ 745
1.7.3.2 Forward Progress ............................. 747
1.8 Transactions [Category: Transactional Memory] .................. 748
1.8.1 Rollback-Only Transactions ..................... 750
1.9 Instruction Storage .............................. 750
1.9.1 Concurrent Modification and Execution of Instructions ............ 752

**Chapter 2. Effect of Operand Placement on Performance**

2.1 Instruction Restart ............................ 753

**Chapter 3. Management of Shared Resources**

3.1 Program Priority Registers ...................... 757
3.2 “or” Instruction ................................ 757

**Chapter 4. Storage Control Instructions**

4.1 Parameters Useful to Application Programs .................... 759
4.2 Data Stream Control Register (DSCR) [Category: Stream] ......... 759
4.3 Cache Management Instructions ...................... 761
4.3.1 Instruction Cache Instructions ................... 762
4.3.2 Data Cache Instructions ......................... 763
4.3.2.1 Obsolete Data Cache Instructions [Category: Vector] .......... 774
4.3.3 “or” Instruction ................................ 774
4.4 Synchronization Instructions ....................... 776
4.4.1 Instruction Synchronize Instruction ................. 776
4.4.2 Load and Reserve and Store Conditional Instructions .......... 776
4.4.2.1 64-Bit Load and Reserve and Store Conditional Instructions [Category: 64-Bit] .................. 782
4.4.2.2 CR0128-bit Load and Reserve Store Conditional Instructions [Category: Load/Store Quadword] .................. 784
4.4.3 Memory Barrier Instructions ...................... 786
4.4.4 Wait Instruction ................................ 791
Chapter 5. Transactional Memory Facility [Category: Transactional Memory] 795

5.1 Transactional Memory Facility Overview 795
5.1.1 Definitions 796
5.2 Transactional Memory Facility States 797
5.2.1 The TDOOMED Bit 799
5.3 Transaction Failure 799
5.3.1 Causes of Transaction Failure 799
5.3.2 Recording of Transaction Failure 801
5.3.3 Handling of Transaction Failure 802
5.4 Transactional Memory Facility Registers 802
5.4.1 Transaction Failure Handler Address Register (TFHAR) 803
5.4.2 Transaction Exception And Summary Register (TEXASR) 803
5.4.3 Transaction Failure Instruction Address Register (TFIAR) 805
5.5 Transactional Facility Instructions 806

Chapter 6. Time Base 813

6.1 Time Base Overview 813
6.2 Time Base 813
6.2.1 Time Base Instructions 813
6.3 Alternate Time Base [Category: Alternate Time Base] 816

Chapter 7. Event-Based Branch Facility [Category: Server] 817

7.1 Event-Based Branch Overview 817
7.2 Event-Based Branch Registers 818
7.2.1 Branch Event Status and Control Register 818
7.2.2 Event-Based Branch Handler Register 819
7.2.3 Event-Based Branch Return Register 819
7.3 Event-Based Branch Instructions 820

Chapter 8. Decorated Storage Facility [Category: Decorated Storage] 821

8.1 Decorated Load Instructions 822
8.2 Decorated Store Instructions 823
8.3 Decorated Notify Instructions 824

Chapter 9. External Control [Category: External Control] 825

9.1 External Access Instructions 826

Appendix A. Assembler Extended Mnemonics 827

A.1 Data Cache Block Flush Mnemonics 827
A.2 Load and Reserve Mnemonics 827
A.3 Synchronize Mnemonics 827
A.4 Wait Mnemonics 827

Appendix B. Programming Examples for Sharing Storage 829

B.1 Atomic Update Primitives 829
B.2 Lock Acquisition and Release, and Related Techniques 831
B.2.1 Lock Acquisition and Import Barriers 831
B.2.1.1 Acquire Lock and Import Shared Storage 831
B.2.1.2 Obtain Pointer and Import Shared Storage 831
B.2.2 Lock Release and Export Barriers 832
B.2.2.1 Export Shared Storage and Release Lock 832
B.2.2.2 Export Shared Storage and Release Lock using Iwsync 832
B.2.3 Safe Fetch 832
B.3 List Insertion 833
B.4 Notes 833
B.5 Transactional Lock Elision [Category: Transactional Memory] 833
B.5.1 Enter Critical Section 834
B.5.2 Handling Busy Lock 834
B.5.3 Handling TLE Abort 834
B.5.4 TLE Exit Section Critical Path 834
B.5.5 Acquisition and Release of TLE Locks 834

Book III-S:

Power ISA Operating Environment Architecture - Server Environment [Category: Server] 837

Chapter 1. Introduction 839

1.1 Overview 839
1.2 Document Conventions 839
1.2.1 Definitions and Notation 839
1.2.2 Reserved Fields 840
1.3 General Systems Overview 840
1.4 Exceptions 841
1.5 Synchronization 841
1.5.1 Context Synchronization 841
1.5.2 Execution Synchronization 842
Chapter 2. Logical Partitioning (LPAR) and Thread Control . . . . . 843
  2.1 Overview ...................................... 843
  2.2 Logical Partitioning Control Register (LPCCR) ............... 843
  2.3 Real Mode Offset Register (RMOR) .......................... 846
  2.4 Hypervisor Real Mode Offset Register (HRMOR) ............... 846
  2.5 Logical Partition Identification Register (LPIDR) .............. 847
  2.6 Processor Compatibility Register (PCR) [Category: Processor Compatibility] 847
  2.7 Other Hypervisor Resources ............................... 853
  2.8 Sharing Hypervisor Resources ............................... 854
  2.9 Sub-Processors ...................................... 854
  2.10 Thread Identification Register (TIR) [Category: Server.Processor Control] 854
  2.11 Hypervisor Interrupt Little-Endian (HILE) Bit ................. 855

Chapter 3. Branch Facility . . . . . . 857
  3.1 Branch Facility Overview ................................ 857
  3.2 Branch Facility Registers ................................. 857
  3.2.1 Machine State Register ............................... 857
  3.2.2 State Transitions Associated with the Transactional Memory Facility [Category: Transactional Memory] 860
  3.3 Branch Facility Instructions ............................... 863
  3.3.1 System Linkage Instructions ......................... 863
  3.3.2 Power-Saving Mode Instructions ........................ 866
  3.3.2.1 Entering and Exiting Power-Saving Mode .............. 869
  3.4 Event-Based Branch Facility and Instruction ................ 870

Chapter 4. Fixed-Point Facility . . . . . . 871
  4.1 Fixed-Point Facility Overview ............................ 871
  4.2 Special Purpose Registers ............................... 871
  4.3 Fixed-Point Facility Registers ............................ 871
  4.3.1 Processor Version Register ............................ 871
  4.3.2 Chip Information Register ............................ 871
  4.3.3 Processor Identification Register ..................... 871
  4.3.4 Control Register ..................................... 872
  4.3.5 Program Priority Register ............................ 872
  4.3.6 Problem State Priority Boost Register ................. 873
  4.3.7 Relative Priority Register ............................ 873
  4.3.8 Software-use SPRs ................................. 873
  4.4 Fixed-Point Facility Instructions ........................ 875
  4.4.1 Fixed-Point Load and Store Caching Inhibited Instructions 875
  4.4.2 OR Instruction ..................................... 878
  4.4.3 Transactional Memory Instructions [Category: Transactional Memory] 879
  4.4.4 Move To/From System Register Instructions .............. 880

Chapter 5. Storage Control . . . . . . 889
  5.1 Overview ......................................... 889
  5.2 Storage Exceptions .................................... 889
  5.3 Instruction Fetch ...................................... 889
  5.3.1 Implicit Branch .................................... 889
  5.3.2 Address Wrapping Combined with Changing MSR Bit SF .... 889
  5.4 Data Access ......................................... 889
  5.5 Performing Operations Out-of-Order ........................ 890
  5.6 Invalid Real Address .................................. 890
  5.7 Storage Addressing .................................... 891
  5.7.1 32-Bit Mode ....................................... 891
  5.7.2 Virtualized Partition Memory (VPM) Mode ............... 891
  5.7.3 Real And Virtual Real Addressing Modes ................. 891
  5.7.3.1 Hypervisor Offset Real Mode Address ................ 892
  5.7.3.2 Offset Real Mode Address .......................... 892
  5.7.3.3 Storage Control Attributes for Accesses in Real and Hypervisor Real Addressing Modes ............... 893
  5.7.3.3.1 Hypervisor Real Mode Storage Control ............. 893
  5.7.3.4 Virtual Real Mode Addressing Mechanism ............. 894
  5.7.3.5 Storage Control Attributes for Implicit Storage Accesses 895
  5.7.4 Address Ranges Having Defined Uses ..................... 895
  5.7.5 Address Translation Overview ........................ 895
  5.7.6 Virtual Address Generation ............................ 896
  5.7.6.1 Segment Lookaside Buffer (SLB) ........................ 896
  5.7.6.2 SLB Search ...................................... 897
  5.7.7 Virtual to Real Translation ............................ 898
  5.7.7.1 Page Table ...................................... 900
  5.7.7.2 Storage Description Register 1 ....................... 901
  5.7.7.3 Page Table Search ................................ 902
  5.7.7.4 Relaxed Page Table Alignment [Category: Server.Relaxed Page Table Alignment] ..................... 904
  5.7.8 Reference and Change Recording ........................ 904
  5.7.9 Storage Protection ................................... 908
  5.7.9.1 Virtual Page Class Key Protection .................... 908
  5.7.9.2 Basic Storage Protection, Address Translation Enabled ........................................... 912
5.7.9.3 Basic Storage Protection, Address Translation Disabled .......... 913
5.8 Storage Control Attributes .......... 914
5.8.1 Guarded Storage .......... 914
5.8.1.1 Out-of-Order Accesses to Guarded Storage .......... 914
5.8.2 Storage Control Bits .......... 914
5.8.2.1 Storage Control Bit Restrictions .......... 915
5.8.2.2 Altering the Storage Control Bits .......... 915
5.9 Storage Control Instructions .......... 917
5.9.1 Cache Management Instructions 917
5.9.2 Synchronize Instruction .......... 917
5.9.3 Lookaside Buffer Management .......... 918
5.9.3.1 SLB Management Instructions 918
5.9.3.2 Bridge to SLB Architecture [Category: Server.Phased-Out] .......... 925
5.9.3.2.1 Segment Register Manipulation Instructions .......... 925
5.9.3.3 TLB Management Instructions 928
5.10 Page Table Update Synchronization Requirements .......... 935
5.10.1 Page Table Updates .......... 935
5.10.1.1 Adding a Page Table Entry .......... 936
5.10.1.2 Modifying a Page Table Entry 937
5.10.1.3 Deleting a Page Table Entry 937

Chapter 6. Interrupts .......... 939
6.1 Overview .......... 939
6.2 Interrupt Registers .......... 939
6.2.1 Machine Status Save/Restore Registers .......... 939
6.2.2 Hypervisor Machine Status Save/Restore Registers .......... 939
6.2.3 Data Address Register .......... 939
6.2.4 Hypervisor Data Address Register .......... 940
6.2.5 Data Storage Interrupt Status Register .......... 940
6.2.6 Hypervisor Data Storage Interrupt Status Register .......... 940
6.2.7 Hypervisor Emulation Instruction Register .......... 940
6.2.8 Hypervisor Maintenance Exception Register .......... 940
6.2.9 Hypervisor Maintenance Exception Enable Register .......... 941
6.2.10 Facility Status and Control Register .......... 941
6.2.11 Hypervisor Facility Status and Control Register .......... 942
6.3 Interrupt Synchronization .......... 946
6.4 Interrupt Classes .......... 946
6.4.1 Precise Interrupt .......... 946
6.4.2 Imprecise Interrupt .......... 946
6.4.3 Interrupt Processing .......... 947
6.4.4 Implicit alteration of HSRR0 and HSRR1 .......... 949
6.5 Interrupt Definitions .......... 950
6.5.1 System Reset Interrupt .......... 952
6.5.2 Machine Check Interrupt .......... 953
6.5.3 Data Storage Interrupt .......... 955
6.5.4 Data Segment Interrupt .......... 956
6.5.5 Instruction Storage Interrupt .......... 957
6.5.6 Instruction Segment Interrupt .......... 957
6.5.7 External Interrupt .......... 957
6.5.8 Alignment Interrupt .......... 957
6.5.9 Program Interrupt .......... 959
6.5.10 Floating-Point Unavailable Interrupt .......... 961
6.5.11 Decrementer Interrupt .......... 961
6.5.12 Hypervisor Decrementer Interrupt .......... 961
6.5.13 Directed Privileged Doorbell Interrupt [Category: Server.Processor Control] .......... 962
6.5.14 System Call Interrupt .......... 962
6.5.15 Trace Interrupt [Category: Trace] .......... 962
6.5.16 Hypervisor Data Storage Interrupt .......... 963
6.5.17 Hypervisor Instruction Storage Interrupt .......... 964
6.5.18 Hypervisor Emulation Assistance Interrupt .......... 965
6.5.19 Hypervisor Maintenance Interrupt .......... 965
6.5.20 Directed Hypervisor Doorbell Interrupt [Category: Server.Processor Control] .......... 966
6.5.22 Vector Unavailable Interrupt [Category: Vector] .......... 966
6.5.23 VSX Unavailable Interrupt [Category: VSX] .......... 966
6.5.24 Facility Unavailable Interrupt .......... 967
6.5.25 Hypervisor Facility Unavailable Interrupt .......... 967
6.6 Partially Executed Instructions .......... 968
6.7 Exception Ordering .......... 969
6.7.1 Unordered Exceptions .......... 969
6.7.2 Ordered Exceptions .......... 969
6.8 Interrupt Priorities .......... 970
6.9 Relationship of Event-Based Branches to Interrupts .......... 972
## Chapter 2. Registers

- 2.2 Registers ........................................... 1027
- 2.2.1 Register Mapping ................................. 1027
- 2.2.2 Logical Partition Identification Register (LPIDR) ........... 1028
- 2.3 Interrupts and Exceptions .......................... 1028
- 2.3.1 Directed Interrupts ............................... 1028
- 2.3.2 Hypervisor Service Interrupts .................... 1029
- 2.4 Instruction Mapping ................................ 1029

## Chapter 3. Thread Control [Category: Embedded Multi-Threading] ...

- 3.1 Overview ............................................ 1031
- 3.2 Thread Identification Register (TIR) .................. 1031
- 3.3 Thread Enable Register (TEN) ....................... 1031
- 3.4 Thread Enable Status Register (TENSR) ............... 1031
- 3.5 Disabling and Enabling Threads....................... 1032
- 3.6 Sharing of Multi-Threaded Processor Resources .......... 1032
- 3.7 Thread Management Facility [Category: Embedded Multithreading:Thread Management] ........ 1033
- 3.7.1 Initialize Next Instruction Address Registers .......... 1033
- 3.7.2 Thread Management Instructions .................. 1034

## Chapter 4. Branch Facility ....... 1035

- 4.1 Branch Facility Overview .......................... 1035
- 4.2 Branch Facility Registers .......................... 1035
- 4.2.1 Machine State Register .......................... 1035
- 4.2.2 Machine State Register Protect Register (MSRP) ....... 1037
- 4.2.3 Embedded Processor Control Register (EPCR) .......... 1038
- 4.3 Branch Facility Instructions ........................ 1040
- 4.3.1 System Linkage Instructions ........................ 1040

## Chapter 5. Fixed-Point Facility . 1045

- 5.1 Fixed-Point Facility Overview ....................... 1045
- 5.2 Special Purpose Registers .......................... 1045
- 5.3 Fixed-Point Facility Registers ....................... 1045
- 5.3.1 Processor Version Register ........................ 1045
- 5.3.2 Chip Information Register ........................ 1045
- 5.3.3 Processor Identification Register .................. 1045
- 5.3.4 Guest Processor Identification Register [Category: Embedded.Hypervisor] .................. 1046
- 5.3.5 Program Priority Register 32-bit .................. 1046
- 5.3.6 Software-use SPRs ................................ 1046
- 5.3.7 External Process ID Registers [Category: Embedded.External PID] .................. 1048

## Chapter 6. Storage Control ....... 1073

- 6.1 Overview ............................................ 1073
- 6.2 Storage Exceptions ................................ 1075
- 6.3 Instruction Fetch ................................... 1075
- 6.3.1 Implicit Branch .................................. 1075
- 6.3.2 Address Wrapping Combined with Changing MSR Bit CM ........ 1076
- 6.4 Data Access ........................................ 1076
- 6.5 Performing Operations Out-of-Order .................. 1076
- 6.6 Invalid Real Address ............................... 1077
- 6.7 Storage Control ..................................... 1077
- 6.7.1 Translation Lookaside Buffer ..................... 1077
- 6.7.2 Virtual Address Spaces ........................... 1081
- 6.7.3 TLB Address Translation ......................... 1082
- 6.7.4 Page Table Address Translation [Category: Embedded.Page Table] ........ 1085
- 6.7.5 Page Table Update Synchronization Requirements [Category: Embedded.Page Table] ........ 1092
- 6.7.5.1 Page Table Updates ............................ 1093
- 6.7.5.1.1 Adding a Page Table Entry .................. 1093
- 6.7.5.1.2 Deleting a Page Table Entry ................. 1094
- 6.7.5.1.3 Modifying a Page Table Entry ............... 1094
- 6.7.5.2 Invalidating an Indirect TLB Entry ................. 1094
- 6.7.6 Storage Access Control ......................... 1094
- 6.7.6.1 Execute Access ............................... 1095
- 6.7.6.2 Write Access .................................. 1095
- 6.7.6.3 Read Access .................................. 1095
- 6.7.6.4 Virtualized Access <E.HV> .......... 1096
- 6.7.6.5 Storage Access Control Applied to Cache Management Instructions ........ 1096
- 6.7.6.6 Storage Access Control Applied to String Instructions .................. 1096
- 6.8 Storage Control Attributes ....................... 1097
- 6.8.1 Guarded Storage ................................ 1097
- 6.8.1.1 Out-of-Order Accesses to Guarded Storage ........ 1097
- 6.8.2 User-Definable .................................. 1097
- 6.8.3 Storage Control Bits ............................. 1097
- 6.8.3.1 Storage Control Bit Restrictions ............... 1098
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8.3.2 Altering the Storage Control Bits</td>
<td>1099</td>
</tr>
<tr>
<td>6.9 Logical to Real Address Translation</td>
<td>1101</td>
</tr>
<tr>
<td>[Category: Embedded.Hypervisor.LRAT]</td>
<td></td>
</tr>
<tr>
<td>6.10 Storage Control Registers</td>
<td>1103</td>
</tr>
<tr>
<td>6.10.1 Process ID Register</td>
<td></td>
</tr>
<tr>
<td>6.10.2 MMU Assist Registers</td>
<td>1103</td>
</tr>
<tr>
<td>6.10.3 MMU Configuration and Control Registers</td>
<td>1104</td>
</tr>
<tr>
<td>6.10.3.1 MMU Configuration Register (MMUCFG)</td>
<td></td>
</tr>
<tr>
<td>6.10.3.2 TLB Configuration Registers (TLBnCFG)</td>
<td></td>
</tr>
<tr>
<td>6.10.3.3 TLB Page Size Registers (TLBnPS) [MAV=2.0]</td>
<td>1106</td>
</tr>
<tr>
<td>6.10.3.4 Embedded Page Table Configuration Register (EPTCFG)</td>
<td></td>
</tr>
<tr>
<td>6.10.3.5 LRAT Configuration Register (LRATCFG) [Category: Embedded.Hypervisor.LRAT]</td>
<td></td>
</tr>
<tr>
<td>6.10.3.6 LRAT Page Size Register (LRATPS) [Category: Embedded.Hypervisor.LRAT]</td>
<td>1107</td>
</tr>
<tr>
<td>6.10.3.7 MMU Control and Status Register (MMUCSR0)</td>
<td>1108</td>
</tr>
<tr>
<td>6.10.3.8 MAS0 Register</td>
<td>1108</td>
</tr>
<tr>
<td>6.10.3.9 MAS1 Register</td>
<td>1108</td>
</tr>
<tr>
<td>6.10.3.10 MAS2 Register</td>
<td>1109</td>
</tr>
<tr>
<td>6.10.3.11 MAS3 Register</td>
<td>1109</td>
</tr>
<tr>
<td>6.10.3.12 MAS4 Register</td>
<td>1110</td>
</tr>
<tr>
<td>6.10.3.13 MAS5 Register</td>
<td>1110</td>
</tr>
<tr>
<td>6.10.3.14 MAS6 Register</td>
<td>1111</td>
</tr>
<tr>
<td>6.10.3.15 MAS7 Register</td>
<td>1111</td>
</tr>
<tr>
<td>6.10.3.16 MAS8 Register [Category: Embedded.Hypervisor]</td>
<td>1112</td>
</tr>
<tr>
<td>6.10.3.17 Accesses to Paired MAS Registers</td>
<td>1112</td>
</tr>
<tr>
<td>6.10.3.18 MAS Register Update Summary</td>
<td>1115</td>
</tr>
<tr>
<td>6.11 Storage Control Instructions</td>
<td>1118</td>
</tr>
<tr>
<td>6.11.1 Cache Management Instructions</td>
<td>1118</td>
</tr>
<tr>
<td>6.11.2 Cache Locking [Category: Embedded Cache Locking]</td>
<td>1119</td>
</tr>
<tr>
<td>6.11.2.1 Lock Setting, Query, and Clearing</td>
<td>1119</td>
</tr>
<tr>
<td>6.11.2.2 Error Conditions</td>
<td>1119</td>
</tr>
<tr>
<td>6.11.2.2.1 Overlocking</td>
<td>1120</td>
</tr>
<tr>
<td>6.11.2.2.2 Unable-to-lock, Unable-to-unlock, and Unable-to-query Conditions</td>
<td>1120</td>
</tr>
<tr>
<td>6.11.2.3 Cache Locking Instructions</td>
<td>1121</td>
</tr>
<tr>
<td>6.11.3 Synchronize Instruction</td>
<td>1124</td>
</tr>
<tr>
<td>6.11.4 LRAT [Category: Embedded.Hypervisor.LRAT] and TLB Management</td>
<td>1124</td>
</tr>
<tr>
<td>6.11.4.1 Reading TLB or LRAT Entries</td>
<td>1124</td>
</tr>
<tr>
<td>6.11.4.2 Writing TLB or LRAT Entries</td>
<td>1125</td>
</tr>
<tr>
<td>6.11.4.2.1 TLB Write Conditional [Category: Embedded.TLB Write Conditional]</td>
<td>1125</td>
</tr>
<tr>
<td>6.11.4.3 Invalidating TLB Entries</td>
<td>1128</td>
</tr>
<tr>
<td>6.11.4.4 TLB Lookaside Information</td>
<td>1129</td>
</tr>
<tr>
<td>6.11.4.5 Invalidating LRAT Entries</td>
<td>1130</td>
</tr>
<tr>
<td>6.11.4.6 Searching TLB Entries</td>
<td>1130</td>
</tr>
<tr>
<td>6.11.4.7 TLB Replacement Hardware Assistance</td>
<td></td>
</tr>
<tr>
<td>6.11.4.8 32-bit and 64-bit Specific MMU Behavior</td>
<td>1131</td>
</tr>
<tr>
<td>6.11.4.9 TLB Management Instructions</td>
<td>1132</td>
</tr>
</tbody>
</table>

## Chapter 7. Interrupts and Exceptions

7.1 Overview                                                        | 1145 |
7.2 Interrupt Registers                                              | 1145 |
7.2.1 Save/Restore Register 0                                         | 1145 |
7.2.2 Save/Restore Register 1                                         | 1145 |
7.2.3 Guest Save/Restore Register 0 [Category: Embedded.Hypervisor] | 1146 |
7.2.4 Guest Save/Restore Register 1 [Category: Embedded.Hypervisor] | 1146 |
7.2.5 Critical Save/Restore Register 0                               | 1147 |
7.2.6 Critical Save/Restore Register 1                               | 1147 |
7.2.7 Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug] | 1147 |
7.2.8 Debug Save/Restore Register 1 [Category: Embedded.Enhanced Debug] | 1148 |
7.2.9 Data Exception Address Register                                | 1148 |
7.2.10 Guest Data Exception Address Register [Category: Embedded.Hypervisor] | 1148 |
7.2.11 Interrupt Vector Prefix Register                              | 1148 |
7.2.12 Guest Interrupt Vector Prefix Register [Category: Embedded.Hypervisor] | 1149 |
7.2.13 Exception Syndrome Register                                   | 1150 |
7.2.14 Guest Exception Syndrome Register [Category: Embedded.Hypervisor] | 1151 |
7.2.15 Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] | 1151 |
7.2.16 Guest Interrupt Vector Offset Register [Category: Embedded.Hypervisor.Phased-Out] | 1152 |
7.2.17 Logical Page Exception Register [Category: Embedded.Hypervisor and Embedded.Page Table] | 1153 |
11.2.2.2 Guest Doorbell Message Filtering [Category: Embedded.Hypervisor] 1231
11.2.2.3 Guest Doorbell Critical Message Filtering [Category: Embedded.Hypervisor] 1232
11.2.2.4 Guest Doorbell Machine Check Message Filtering [Category: Embedded.Hypervisor] 1232
11.3 Processor Control Instructions 1233

Chapter 12. Synchronization
Requirements for Context Alterations 1235

Appendix
A. Implementation-Dependent Instructions 1239
   A.1 Embedded Cache Initialization [Category: Embedded.Cache Initialization] 1239
   A.2 Embedded Cache Debug Facility [Category: Embedded.Cache Debug] 1240
   A.2.1 Embedded Cache Debug Registers 1240
      A.2.1.1 Data Cache Debug Tag Register High 1240
      A.2.1.2 Data Cache Debug Tag Register Low 1240
      A.2.1.3 Instruction Cache Debug Data Register 1241
      A.2.1.4 Instruction Cache Debug Tag Register High 1241
      A.2.1.5 Instruction Cache Debug Tag Register Low 1241
      A.2.2 Embedded Cache Debug Instructions 1242

Appendix B. Assembler Extended Mnemonics 1245
   B.1 Move To/From Special Purpose Register Mnemonics 1246
   B.2 Data Cache Block Flush Mnemonics [Category: Embedded.Phased In] 1247

Appendix C. Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations 1249
   C.1 Hardware Guidelines 1249
      C.1.1 64-bit Specific Instructions 1249
      C.1.2 Registers on 32-bit Implementations 1249
      C.1.3 Addressing on 32-bit Implementations 1249
   C.1.4 TLB Fields on 32-bit Implementations 1249
   C.1.5 Thread Control and Status on 32-bit Implementations 1249
   C.2 32-bit Software Guidelines 1249
      C.2.1 32-bit Instruction Selection 1249

Appendix D. Example Performance Monitor [Category: Embedded.Performance Monitor] 1251
   D.1 Overview 1251
   D.2 Programming Model 1251
      D.2.1 Event Counting 1252
      D.2.2 Thread Context Configurability 1252
      D.2.3 Event Selection 1252
      D.2.4 Thresholds 1253
      D.2.5 Performance Monitor Exception 1253
      D.2.6 Performance Monitor Interrupt 1253
      D.3 Performance Monitor Registers 1253
      D.3.1 Performance Monitor Global Control Register 0 1253
      D.3.2 Performance Monitor Local Control A Registers 1254
      D.3.3 Performance Monitor Local Control B Registers 1255
      D.3.4 Performance Monitor Counter Registers 1255
      D.4 Performance Monitor Instructions 1257
      D.5 Performance Monitor Software Usage Notes 1258
      D.5.1 Chaining Counters 1258
      D.5.2 Thresholding 1258

Book VLE:

Chapter 1. Variable Length Encoding Introduction 1261
   1.1 Overview 1261
   1.2 Documentation Conventions 1261
      1.2.1 Description of Instruction Operation 1261
   1.3 Instruction Mnemonics and Operands 1261
      1.4 VLE Instruction Formats 1262
Appendix A. VLE Instruction Set
Sorted by Mnemonic .......................... 1315

Appendix B. VLE Instruction Set
Sorted by Opcode ............................ 1331

Appendices:

Power ISA Book I-III Appendices 1347

Appendix A. Incompatibilities with
the POWER Architecture ................. 1349
  A.1 New Instructions, Formerly Privileged
      Instructions ............................ 1349
  A.2 Newly Privileged
      Instructions ............................ 1349
  A.3 Reserved Fields in
      Instructions ............................ 1349
  A.4 Reserved Bits in Registers .......... 1349
  A.5 Alignment Check ........................ 1349
  A.6 Condition Register .................... 1350
  A.7 LK and Rc Bits ........................ 1350
  A.8 BO Field ............................... 1350
  A.9 BH Field ............................... 1350
  A.10 Branch Conditional to Count Register
       ........................................ 1350
  A.11 System Call ........................... 1350
  A.12 Fixed-Point Exception
      Register (XER) ......................... 1351
  A.13 Update Forms of Storage Access
      Instructions ........................... 1351
  A.14 Multiple Register Loads .......... 1351
  A.15 Load/Store Multiple Instructions 1351
  A.16 Move Assist Instructions ........... 1351
  A.17 Move To/From SPR .................... 1351
  A.18 Effects of Exceptions on FPSCR Bits
      FR and FI ............................... 1352
  A.19 Store Floating-Point Single Instruc-
      tions ................................. 1352
  A.20 Move From FPSCR ..................... 1352
  A.21 Zeroing Bytes in the Data Cache ....
       ........................................ 1352
  A.22 Synchronization ...................... 1352
  A.23 Move To Machine State Register
      Instruction ............................ 1352
  A.24 Direct-Store Segments ............... 1352
  A.25 Segment Register
      Manipulation Instructions ............. 1352
  A.26 TLB Entry Invalidation .......... 1353
  A.27 Alignment Interrupts ............... 1353
  A.28 Floating-Point Interrupts .......... 1353
  A.29 Timing Facilities .................... 1353
  A.29.1 Real-Time Clock ................... 1353
  A.29.2 Decrementer ....................... 1353
  A.30 Deleted Instructions ............... 1354
  A.31 Discontinued Opcodes ............... 1354
  A.32 POWER2 Compatibility ............... 1355
  A.32.1 Cross-Reference for Changed
       POWER2 Mnemonics .................... 1355
  A.32.2 Load/Store Floating-Point Double...
       ........................................ 1355
  A.32.3 Floating-Point Conversion to Inte-
       ger ................................. 1355
  A.32.4 Floating-Point Interrupts ....... 1356
  A.32.5 Trace .............................. 1356
  A.33 Deleted Instructions ............... 1356
  A.33.1 Discontinued Opcodes ............. 1356

Appendix B. Platform Support
Requirements ............................... 1357

Appendix C. Complete SPR List 1361

Appendix D. Illegal Instructions 1367

Appendix E. Reserved Instructions .. 1369

Appendix F. Opcode Maps ............... 1371

Appendix G. Power ISA Instruction
Set Sorted by Category ..................... 1395

Appendix H. Power ISA Instruction
Set Sorted by Opcode ....................... 1425

Appendix I. Power ISA Instruction
Set Sorted by Mnemonic .................... 1455

Index ........................................ 1485

Last Page - End of Document ... 1495

xxiv  Power ISA™ - Book I
Figures

Preface ....................................................... iii
Table of Contents ................................. vii
Figures ........................................... xxv

Book I:

Power ISA User Instruction Set Architec-

ture .......................................................... 1

1. Category Listing ........................................ 8
2. Logical processing model ............................. 11
3. Registers that are defined in Book I ............ 12
4. I instruction format .................................. 14
5. B instruction format .................................. 14
6. SC instruction format ................................ 14
7. D instruction format .................................. 14
8. DS instruction format ................................ 14
9. DQ instruction format ................................ 14
10. X Instruction Format .................................. 15
11. XL instruction format ................................ 15
12. XFX instruction format ............................... 15
13. XFL instruction format ............................... 16
14. XX1 Instruction Format ............................... 16
15. XX2 Instruction Format ............................... 16
16. XX3 Instruction Format ............................... 16
17. XX4-Form Instruction Format ....................... 16
18. XS instruction format ................................ 16
19. XO instruction format ................................ 16
20. A instruction format .................................. 16
21. M instruction format .................................. 16
22. MD instruction format ................................ 16
23. MDS instruction format ............................. 16
24. VA instruction format ................................ 16
25. VC instruction format ................................ 16
26. VX instruction format ................................ 17
27. EVX instruction format .............................. 17
28. EVS instruction format .............................. 17
29. Z22 instruction format .............................. 17
30. Z23 instruction format .............................. 17
31. Storage operands and byte ordering ............ 23
32. C structure ‘s’, showing values of elements .... 24
33. Big-Endian mapping of structure ‘s’ ............ 24
34. Little-Endian mapping of structure ‘s’ .......... 24
35. Instructions and byte ordering .................... 24
36. Assembly language program ‘p’ .................... 24
37. Big-Endian mapping of program ‘p’ ............ 25
38. Little-Endian mapping of program ‘p’ .......... 25
39. Condition Register ................................. 30
40. Link Register ........................................... 32
41. Count Register ........................................ 32
42. Target Address Register ............................ 32
43. Branch History Rolling Buffer Data ............ 33
44. BO field encodings .................................. 34
45. "at" bit encodings .................................... 34
46. BH field encodings .................................. 35
47. General Purpose Registers ......................... 45
48. Fixed-Point Exception Register ................. 45
49. Software-use SPRs .................................... 46
50. Floating-Point Registers ............................ 116
51. Floating-Point Status and Control Register .......................... 116
52. Floating-Point Result Flags ......................... 119
53. Floating-point single format ...................... 119
54. Floating-point double format ..................... 119
55. IEEE floating-point fields ......................... 119
56. Approximation to real numbers .................. 119
57. Selection of Z1 and Z2 .............................. 123
58. IEEE 64-bit execution model ....................... 129
59. Interpretation of G, R, and X bits ............... 129
60. Location of the Guard, Round, and Sticky bits in the IEEE execution model .......................... 129
61. Multiply-add 64-bit execution model ............ 131
62. Location of the Guard, Round, and Sticky bits in the multiply-add execution model .................. 131
63. Vector Register elements ........................... 171
64. Vector Registers ..................................... 172
65. Vector Status and Control Register ............. 172
66. Aligned quadword storage operand ............... 174
67. Vector Register contents for aligned quadword Load or Store ................................. 174
68. Unaligned quadword storage operand .......... 174
69. Vector Register contents ........................... 174
70. Vector Gather Bits by Bytes by Doubleword .... 262
72. Format for Unsigned Decimal Data .............. 272
73. Format for Signed Decimal Data .................. 272
74. Summary of BCD Digit and Sign Codes ........ 273
75. DFP Short format ..................................... 273
76. DFP Long format ..................................... 273
77. DFP Extended format ............................... 273
78. Encoding of the G field for Special Symbols ... 274
79. Encoding of bits 0:4 of the G field for Finite Numbers
<table>
<thead>
<tr>
<th>51. MAS register</th>
<th>1114</th>
</tr>
</thead>
<tbody>
<tr>
<td>52. MAS8 register</td>
<td>1114</td>
</tr>
<tr>
<td>53. Save/Restore Register 0</td>
<td>1145</td>
</tr>
<tr>
<td>54. Save/Restore Register 1</td>
<td>1145</td>
</tr>
<tr>
<td>55. Guest Save/Restore Register 0</td>
<td>1146</td>
</tr>
<tr>
<td>56. Guest Save/Restore Register 1</td>
<td>1146</td>
</tr>
<tr>
<td>57. Critical Save/Restore Register 0</td>
<td>1147</td>
</tr>
<tr>
<td>58. Critical Save/Restore Register 1</td>
<td>1147</td>
</tr>
<tr>
<td>59. Debug Save/Restore Register 0</td>
<td>1147</td>
</tr>
<tr>
<td>60. Debug Save/Restore Register 1</td>
<td>1148</td>
</tr>
<tr>
<td>61. Exception Syndrome Register Definitions</td>
<td>1151</td>
</tr>
<tr>
<td>62. Interrupt Vector Offset Register Assignments</td>
<td>1152</td>
</tr>
<tr>
<td>63. Guest Interrupt Vector Offset Register Assignments</td>
<td>1152</td>
</tr>
<tr>
<td>64. Logical Page Exception Register</td>
<td>1153</td>
</tr>
<tr>
<td>65. Machine Check Save/Restore Register 0</td>
<td>1153</td>
</tr>
<tr>
<td>66. Machine Check Save/Restore Register 1</td>
<td>1154</td>
</tr>
<tr>
<td>67. External Proxy Register</td>
<td>1154</td>
</tr>
<tr>
<td>68. Guest External Proxy Register</td>
<td>1155</td>
</tr>
<tr>
<td>69. Interrupt and Exception Types</td>
<td>1164</td>
</tr>
<tr>
<td>70. Interrupt Vector Offsets</td>
<td>1165</td>
</tr>
<tr>
<td>71. Interrupt Hierarchy</td>
<td>1187</td>
</tr>
<tr>
<td>72. Machine State Register Initial Values</td>
<td>1193</td>
</tr>
<tr>
<td>73. TLB Initial Values</td>
<td>1194</td>
</tr>
<tr>
<td>74. Time Base</td>
<td>1197</td>
</tr>
<tr>
<td>75. Decrementer</td>
<td>1199</td>
</tr>
<tr>
<td>76. Guest Decrementer</td>
<td>1200</td>
</tr>
<tr>
<td>77. Decrementer Auto-Reload Register</td>
<td>1201</td>
</tr>
<tr>
<td>78. Guest Decrementer Auto-Reload Register</td>
<td>1201</td>
</tr>
<tr>
<td>79. Relationships of the Timer Facilities</td>
<td>1203</td>
</tr>
<tr>
<td>80. Relationships of the Guest Timer Facilities</td>
<td>1206</td>
</tr>
<tr>
<td>81. Guest Timer Status Register Write Register</td>
<td>1208</td>
</tr>
<tr>
<td>82. Watchdog State Machine</td>
<td>1209</td>
</tr>
</tbody>
</table>

83. Watchdog Timer Controls
84. Guest Watchdog State Machine
85. Guest Watchdog Timer Controls
86. Debug Status Register Write Register
87. Data Cache Debug Tag Register High
88. Data Cache Debug Tag Register Low
89. Instruction Cache Debug Data Register
90. Instruction Cache Debug Tag Register High
91. Instruction Cache Debug Tag Register Low
92. Thread States and PMLCan Bit Settings
93. [User] Performance Monitor Global Control Register 0
94. [User] Performance Monitor Local Control A Registers
95. [User] Performance Monitor Local Control B Register
96. [User] Performance Monitor Counter Registers
97. Embedded Performance Monitor PMRs

Book VLE:

Power ISA Operating Environment Architecture - Variable Length Encoding (VLE) Environment
[Category: Variable Length Encoding]...

1. BD8 instruction format
2. C instruction format
3. IM5 instruction format
4. OIM5 instruction format
5. IM7 instruction format
6. R instruction format
7. RR instruction format
8. SD4 instruction format
9. BD15 instruction format
10. BD24 instruction format
11. D8 instruction format
12. I16A instruction format
13. I16L instruction format
14. M instruction format
15. SC18 instruction format
16. LI20 instruction format
17. X instruction format
18. Condition Register
19. BO32 field encodings
20. BO16 field encodings

Appendices:

Power ISA Book I-III Appendices...
Book I:

Power ISA User Instruction Set Architecture
Chapter 1. Introduction

1.1 Overview
This chapter describes computation modes, document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Instruction Mnemonics and Operands
The description of each instruction includes the mnemonic and a formatted list of operands. Some examples are the following.

- `stw RS,D(RA)`
- `addis RT,RA,SI`

Power ISA-compliant Assemblers will support the mnemonics and operand lists exactly as shown. They should also provide certain extended mnemonics, such as the ones described in Appendix E of Book I.

1.3 Document Conventions

1.3.1 Definitions
The following definitions are used throughout this document.

- **program**
  A sequence of related instructions.

- **application program**
  A program that uses only the instructions and resources described in Books I and II.

- **processor**
  The hardware component that implements the instruction set, storage model, and other facilities defined in the Power ISA architecture, and executes the instructions specified in a program.

- **quadwords, doublewords, words, halfwords, and bytes**
  128 bits, 64 bits, 32 bits, 16 bits, and 8 bits, respectively.

- **positive**
  Means greater than zero.

- **negative**
  Means less than zero.

- **floating-point single format** (or simply single format)
  Refers to the representation of a single-precision binary floating-point value in a register or storage.

- **floating-point double format** (or simply double format)
  Refers to the representation of a double-precision binary floating-point value in a register or storage.

- **system library program**
  A component of the system software that can be called by an application program using a Branch instruction.

- **system service program**
  A component of the system software that can be called by an application program using a System Call instruction.

- **system trap handler**
  A component of the system software that receives control when the conditions specified in a Trap instruction are satisfied.

- **system error handler**
  A component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.

- **latency**
  Refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.

- **unavailable**
  Refers to a resource that cannot be used by the program. For example, storage is unavailable if access to it is denied. See Book III.
undefined value
May vary between implementations, and between different executions on the same implementation, and similarly for register contents, storage contents, etc., that are specified as being undefined.

boundedly undefined
The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary finite sequence of instructions (none of which yields boundedly undefined results) in the state the processor was in before executing the given instruction. Boundedly undefined results may include the presentation of inconsistent state to the system error handler as described in Section 1.9.1 of Book II. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation.

“must”
If software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), the results are boundedly undefined unless otherwise stated.

sequential execution model
The model of program execution described in Section 2.2, “Instruction Execution Order” on page 29.

Auxiliary Processor
An implementation-specific processing unit. Previous versions of the architecture use the term Auxiliary Processing Unit (APU) to describe this extension of the architecture. Architectural support for auxiliary processors is part of the Embedded category.

virtualized implementation
An implementation of the Power Architecture created by hypervisor software. A guest operating system sees a virtualized implementation of the Power ISA. Architectural support for virtualized implementations is part of the Embedded category (see Section 1.3.5, “Categories”).

1.3.2 Notation
The following notation is used throughout the Power ISA documents.

- All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.
  Underscores may be used between digits.
- RT, RA, R1, ... refer to General Purpose Registers.
- FRT, FRA, FR1, ... refer to Floating-Point Registers.
- FRTp, FRAp, FRBp, ... refer to an even-odd pair of Floating-Point Registers. Values must be even, otherwise the instruction form is invalid.
- VRT, VRA, VR1, ... refer to Vector Registers.
- (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Parentheses are also omitted when register x is the register into which the result of an operation is placed.
- (RA/0) means the contents of register RA if the RA field has the value 1-31, or the value 0 if the RA field is 0.
- Bytes in instructions, fields, and bit strings are numbered from left to right, starting with byte 0 (most significant).
- Bits in registers, instructions, fields, and bit strings are specified as follows. In the last three items (definition of Xp etc.), if X is a field that specifies a GPR, FPR, or VR (e.g., the RS field of an instruction), the definitions apply to the register, not to the field.
  - Bits in instructions, fields, and bit strings are numbered from left to right, starting with bit 0
  - For all registers except the Vector category, bits in registers that are less than 64 bits start with bit number 64-L, where L is the register length; for the Vector category, bits in registers that are less than 128 bits start with bit number 128-L.
  - The leftmost bit of a sequence of bits is the most significant bit of the sequence.
  - Xp means bit p of register/instruction/field/bit_string X.
  - Xp:q means bits p through q of register/instruction/field/bit_string X.
  - Xp q ... means bits p, q, ... of register/instruction/field/bit_string X.
- ¬(RA) means the one’s complement of the contents of register RA.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution.
- The symbol ‖ is used to describe the concatenation of two values. For example, 010 ‖ 111 is the same as 010111.
- x^n means x raised to the n^th power.
nx means the replication of x, n times (i.e., x concatenated to itself n–1 times). n0 and n1 are special cases:
- n0 means a field of n bits with each bit equal to 0. Thus n0 is equivalent to 0b00000.
- n1 means a field of n bits with each bit equal to 1. Thus n1 is equivalent to 0b11111.

Each bit and field in instructions, and in status and control registers (e.g., XER, FPSCR) and Special Purpose Registers, is either defined or reserved. Some defined fields contain reserved values. In such cases when this document refers to the specific field, it refers only to the defined values, unless otherwise specified.

/, //, ///, ... denotes a reserved field, in a register, instruction, field, or bit string.

?, ??, ???, ... denotes an implementation-dependent field in a register, instruction, field or bit string.

1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs

Reserved fields in instructions are ignored by the processor. This is a requirement in the Server environment and is being phased into the Embedded environment.

In some cases a defined field of an instruction has certain values that are reserved. This includes cases in which the field is shown in the instruction layout as containing a particular value; in such cases all other values of the field are reserved. In general, if an instruction is coded such that a defined field contains a reserved value the instruction form is invalid; see Section 1.8.2 on page 22. The only exceptions to the preceding rule is that it does not apply to Reserved and Illegal classes of instructions (see Section 1.7) or to portions of defined fields that are specified, in the instruction description, as being treated as reserved fields.

To maximize compatibility with future architecture extensions, software must ensure that reserved fields in instructions contain zero and that defined fields of instructions do not contain reserved values.

The handling of reserved bits in System Registers (e.g., XER, FPSCR) depends on whether the processor is in problem state. Unless otherwise stated, software is permitted to write any value to such a bit. In problem state, a subsequent reading of the bit returns 0 regardless of the value written; in privileged states, a subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

In some cases, a defined field of a System Register has certain values that are reserved. Software must not set a defined field of a System Register to a reserved value. References elsewhere in this document to a defined field (in an instruction or System Register) that has reserved values assume the field does not contain a reserved value, unless otherwise stated or obvious from context.

In some cases, a given bit of a System Register is specified to be set to a constant value by a given instruction or event. Unless otherwise stated or obvious from context, software should not depend on this constant value because the bit may be assigned a meaning in a future version of the architecture.

The reserved SPRs include SPRs 808, 809, 810, and 811. mtspr and mfspr instructions specifying these SPRs are treated as noops. Reserved SPRs are provided in the architecture to anticipate the eventual adoption of performance hint functionality that must be controlled by SPRs. Control of these capabilities using reserved SPRs will allow software to use these new capabilities on new implementations that support them while remaining compatible with existing implementations that may not support the new functionality.
Reserved SPRs are not assigned names. There are no individual descriptions of reserved SPRs in this document.

**Assembler Note**

Assemblers should report uses of reserved values of defined fields of instructions as errors.

**Programming Note**

It is the responsibility of software to preserve bits that are now reserved in System Registers, because they may be assigned a meaning in some future version of the architecture.

In order to accomplish this preservation in implementation-independent fashion, software should do the following.

- Initialize each such register supplying zeros for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

The XER and FPSCR are partial exceptions to this recommendation. Software can alter the status bits in these registers, preserving the reserved bits, by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined bit in the FPSCR by executing a Floating-Point Status and Control Register instruction. Using such instructions is likely to yield better performance than using the method described in the second item above.

### 1.3.4 Description of Instruction Operation

Instruction descriptions (including related material such as the introduction to the section describing the instructions) mention that the instruction may cause a system error handler to be invoked, under certain conditions, if and only if the system error handler may treat the case as a programming error. (An instruction may cause a system error handler to be invoked under other conditions as well; see Chapter 6 of Book III-S and Chapter 7 of Book III-E).

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the notation described in Section 1.3.2. Some of this notation is also used in the formal descriptions of instructions. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instruction, except that “standard” setting of status registers, such as the Condition Register, is not shown. (“Non-standard” setting of these registers, such as the setting of the Condition Register by the Compare instructions, is shown.) The RTL descriptions do not cover cases in which the system error handler is invoked, or for which the results are boundedly undefined.

The RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment</td>
</tr>
<tr>
<td>←iea</td>
<td>Assignment of an instruction effective address. In 32-bit mode the high-order 32 bits of the 64-bit target address are set to 0.</td>
</tr>
<tr>
<td>¬</td>
<td>NOT logical operator</td>
</tr>
<tr>
<td>+</td>
<td>Two’s complement addition</td>
</tr>
<tr>
<td>-</td>
<td>Two’s complement subtraction, unary minus</td>
</tr>
<tr>
<td>×</td>
<td>Multiplication</td>
</tr>
<tr>
<td>×si</td>
<td>Signed-integer multiplication</td>
</tr>
<tr>
<td>×ui</td>
<td>Unsigned-integer multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>÷</td>
<td>Division, with result truncated to integer</td>
</tr>
<tr>
<td>√</td>
<td>Square root</td>
</tr>
<tr>
<td>=, ≠</td>
<td>Equals, Not Equals relations</td>
</tr>
<tr>
<td>&lt;, ≤, &gt;, ≥</td>
<td>Signed comparison relations</td>
</tr>
<tr>
<td>&lt;u, &gt;u</td>
<td>Unsigned comparison relations</td>
</tr>
<tr>
<td>?</td>
<td>Unordered comparison relation</td>
</tr>
<tr>
<td>&amp;,</td>
<td>AND, OR logical operators</td>
</tr>
<tr>
<td>⊕, ≡</td>
<td>Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))</td>
</tr>
<tr>
<td>ABS(x)</td>
<td>Absolute value of x</td>
</tr>
<tr>
<td>BCD_TO_DPD(x)</td>
<td>The low-order 24 bits of x contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the result. See Section B.1, “BCD-to-DPD Translation”.</td>
</tr>
<tr>
<td>CEIL(x)</td>
<td>Least integer ≥ x</td>
</tr>
<tr>
<td>DCR(x)</td>
<td>Device Control Register x &lt;E.DC&gt;</td>
</tr>
<tr>
<td>DOUBLE(x)</td>
<td>Result of converting x from floating-point single format to floating-point double format, using the model shown on page 133</td>
</tr>
<tr>
<td>DPD_TO_BCD(x)</td>
<td>The low-order 20 bits of x contain two declets which are converted to six, 4-bit BCD fields; each set of six 4-bit BCD fields is placed into the low-order 24 bits of the result. See Section B.2, “DPD-to-BCD Translation”.</td>
</tr>
<tr>
<td>EXTS(x)</td>
<td>Result of extending x on the left with sign bits</td>
</tr>
<tr>
<td>FLOOR(x)</td>
<td>Greatest integer ≤ x</td>
</tr>
<tr>
<td>GPR(x)</td>
<td>General Purpose Register x</td>
</tr>
</tbody>
</table>
MASK(x, y) Mask having 1s in positions x through y (wrapping if x > y) and 0s elsewhere

MEM(x, y) Contents of a sequence of y bytes of storage. The sequence depends on the byte ordering used for storage access, as follows.
  Big-Endian byte ordering:
  The sequence starts with the byte at address x and ends with the byte at address x+y-1.
  Little-Endian byte ordering:
  The sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MEM_DECORATED(x, y, z)
Contents of a sequence of y bytes of storage, where the storage is accessed with decoration z applied. The sequence depends on the byte ordering used for storage access, as follows.
  Big-Endian byte ordering:
  The sequence starts with the byte at address x and ends with the byte at address x+y-1.
  Little-Endian byte ordering:
  The sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MEM_NOTIFY(x, z)
The decoration z is sent to storage location x.

ROTL64(x, y) Result of rotating the 64-bit value x left y positions

ROTL32(x, y) Result of rotating the 64-bit value x||x left y positions, where x is 32 bits long

SINGLE(x) Result of converting x from floating-point double format to floating-point single format, using the model shown on page 137

SPR(x) Special Purpose Register x

switch/case/default statement, indenting shows range. The clause after “switch” specifies the expression to evaluate. The clause after “case” specifies individual values for the expression, followed by a colon, followed by the actions that are taken if the evaluated expression has any of the specified values. “default” is optional. If present, it must follow all the “case” clauses. The clause after “default” starts with a colon, and specifies the actions that are taken if the evaluated expression does not have any of the values specified in the preceding case statements.

TRAP Invoke the system trap handler

Characterization
Reference to the setting of status bits, in a standard way that is explained in the text

undefined An undefined value.
The precedence rules for RTL operators are summarized in Table 1. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, \(-\) associates from left to right, so \(a-b-c = (a-b)-c\).)

Parentheses are used to override the evaluation order implied by the table or to increase clarity; parenthesized expressions are evaluated before serving as operands.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscript, function evaluation</td>
<td>left to right</td>
</tr>
<tr>
<td>pre-superscript (replication),</td>
<td>right to left</td>
</tr>
<tr>
<td>post-superscript (exponentiation)</td>
<td></td>
</tr>
<tr>
<td>unary (-), (\sim)</td>
<td>right to left</td>
</tr>
<tr>
<td>(\times), (\div)</td>
<td>left to right</td>
</tr>
<tr>
<td>(+), (-)</td>
<td>left to right</td>
</tr>
<tr>
<td>(|)</td>
<td>left to right</td>
</tr>
<tr>
<td>(=), (\neq), (&lt;), (\leq), (&gt;), (\geq), (&lt;^u), (&gt;^u), (?)</td>
<td>left to right</td>
</tr>
</tbody>
</table>

### 1.3.5 Categories

Each facility (including registers and fields therein) and instruction is in exactly one of the categories listed in Figure 1.

A category may be defined as a dependent category. These are categories that are supported only if the category they are dependent on is also supported. Dependent categories are identified by the “*” in their category name, e.g., if an implementation supports the Floating-Point.Record category, then the Floating-Point category is also supported.

An implementation that supports a facility or instruction in a given category, except for the two categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Abvr.</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>B</td>
<td>Required for all implementations</td>
</tr>
<tr>
<td>Server</td>
<td>S</td>
<td>Required for Server implementations</td>
</tr>
<tr>
<td>Embedded</td>
<td>E</td>
<td>Required for Embedded implementations</td>
</tr>
<tr>
<td>Alternate Time Base</td>
<td>ATB</td>
<td>An additional Time Base; see Book II</td>
</tr>
<tr>
<td>Cache Specification</td>
<td>CS</td>
<td>Specify a specific cache for some instructions; see Book II</td>
</tr>
<tr>
<td>Decimal Floating-Point</td>
<td>DFP</td>
<td>Decimal Floating-Point facilities</td>
</tr>
<tr>
<td>Decorated Storage</td>
<td>DS</td>
<td>Decorated Storage facilities</td>
</tr>
<tr>
<td>Elemental Memory Barriers</td>
<td>EMB</td>
<td>More granular memory barrier support</td>
</tr>
<tr>
<td>Embedded.Cache Debug</td>
<td>E.CD</td>
<td>Provides direct access to cache data and directory content</td>
</tr>
<tr>
<td>Embedded.Cache Initialization</td>
<td>E.CI</td>
<td>Instructions that invalidate the entire cache</td>
</tr>
<tr>
<td>Embedded.Device Control</td>
<td>E.DC</td>
<td>Legacy Device Control bus support</td>
</tr>
<tr>
<td>Embedded.Enhanced Debug</td>
<td>E.ED</td>
<td>Embedded Enhanced Debug facility; see Book III-E</td>
</tr>
<tr>
<td>Embedded.External PID</td>
<td>E.PD</td>
<td>Embedded External PID facility; see Book III-E</td>
</tr>
<tr>
<td>Embedded.Hypervisor</td>
<td>E.HV</td>
<td>Embedded Logical Partitioning and hypervisor facilities</td>
</tr>
<tr>
<td>Embedded.Hypervisor.LRAT</td>
<td>E.HVLRAT</td>
<td>Embedded Hypervisor Logical to Real Address Translation facility; see Book III-E</td>
</tr>
<tr>
<td>Embedded.Little-Endian</td>
<td>E.LE</td>
<td>Embedded Little-Endian page attribute; see Book III-E</td>
</tr>
<tr>
<td>Embedded.MMU Type FSL</td>
<td>E.MMUF</td>
<td>Type FSL Storage Control</td>
</tr>
<tr>
<td>Embedded.Page Table</td>
<td>E.PT</td>
<td>Embedded Page Table facility; see Book III-E</td>
</tr>
<tr>
<td>Embedded.TLB Write Conditional</td>
<td>E.TWC</td>
<td>Embedded TLB Write Conditional facility; see Book III-E</td>
</tr>
<tr>
<td>Embedded.Performance Monitor</td>
<td>E.PM</td>
<td>Embedded Performance Monitor example; see Book III-E</td>
</tr>
<tr>
<td>Embedded.Processor Control</td>
<td>E.PC</td>
<td>Embedded Processor Control facility; see Book III-E</td>
</tr>
<tr>
<td>Embedded Cache Locking</td>
<td>ECL</td>
<td>Embedded Cache Locking facility; see Book III-E</td>
</tr>
</tbody>
</table>

Figure 1. Category Listing (Sheet 1 of 2)
<table>
<thead>
<tr>
<th>Category</th>
<th>Abvr.</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded Multi-Threading</td>
<td>EM</td>
<td>Embedded Multi-Threading; see Book III-E</td>
</tr>
<tr>
<td>Embedded Multi-Threading. Thread Management</td>
<td>EM.TM</td>
<td>Embedded Multi-Threading Thread Management Facility</td>
</tr>
<tr>
<td>External Control</td>
<td>EC</td>
<td>External Control facility; see Book II</td>
</tr>
<tr>
<td>External Proxy</td>
<td>EXP</td>
<td>External Proxy facility; see Book III-E</td>
</tr>
<tr>
<td>Floating-Point</td>
<td>FP</td>
<td>Floating-Point Facilities</td>
</tr>
<tr>
<td>Floating-Point.Record</td>
<td>FPR</td>
<td>Floating-Point instructions with Rc=1</td>
</tr>
<tr>
<td>Legacy Integer Multiply-Accumulate¹</td>
<td>LMA</td>
<td>Legacy Integer Multiply-accumulate instructions</td>
</tr>
<tr>
<td>Legacy Move Assist</td>
<td>LMV</td>
<td>Determine Left most Zero Byte instruction</td>
</tr>
<tr>
<td>Load/Store Quadword</td>
<td>LSQ</td>
<td>Load/Store Quadword instructions; see Book III-S</td>
</tr>
<tr>
<td>Memory Coherence</td>
<td>MMC</td>
<td>Requirement for Memory Coherence; see Book II</td>
</tr>
<tr>
<td>Move Assist</td>
<td>MA</td>
<td>Move Assist instructions</td>
</tr>
<tr>
<td>Processor Compatibility</td>
<td>PCR</td>
<td>Processor Compatibility Register</td>
</tr>
<tr>
<td>Server.Performance Monitor</td>
<td>S.PM</td>
<td>Server Performance Monitor example; see Book III-S</td>
</tr>
<tr>
<td>Server.Processor Control</td>
<td>S.PC</td>
<td>Server Processor Control Facility; see Book III-S</td>
</tr>
<tr>
<td>Server.Relaxed Page Table Alignment</td>
<td>S.RPTA</td>
<td>HTAB alignment on 256 KB boundary; see Book III-S</td>
</tr>
<tr>
<td>Signal Processing Engine¹,²</td>
<td>SP</td>
<td>Facility for signal processing</td>
</tr>
<tr>
<td>SPE.Embedded Float Scalar Double</td>
<td>SPFD</td>
<td>GPR-based Floating-Point double-precision instruction set</td>
</tr>
<tr>
<td>SPE.Embedded Float Scalar Single</td>
<td>SPFS</td>
<td>GPR-based Floating-Point single-precision instruction set</td>
</tr>
<tr>
<td>SPE.Embedded Float Vector</td>
<td>SPFV</td>
<td>GPR-based Floating-Point Vector instruction set</td>
</tr>
<tr>
<td>Store Conditional Page Mobility</td>
<td>SCPM</td>
<td>Store Conditional accounting for page movement; see Book II</td>
</tr>
<tr>
<td>Stream</td>
<td>STM</td>
<td>Stream variant of dcbt instruction; see Book II</td>
</tr>
<tr>
<td>Strong Access Order</td>
<td>SAO</td>
<td>Assist for X86 and Sparc emulation; see Book II</td>
</tr>
<tr>
<td>Trace</td>
<td>TRC</td>
<td>Trace Facility; see Book III-S</td>
</tr>
<tr>
<td>Transactional Memory</td>
<td>TM</td>
<td>Full hardware Transactional Memory support</td>
</tr>
<tr>
<td>Variable Length Encoding</td>
<td>VLE</td>
<td>Variable Length Encoding facility; see Book VLE</td>
</tr>
<tr>
<td>Vector-Scalar Extension</td>
<td>VSX</td>
<td>Vector-Scalar Extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires implementation of Floating-Point and Vector categories</td>
</tr>
<tr>
<td>Vector¹</td>
<td>V</td>
<td>Vector facilities</td>
</tr>
<tr>
<td>Vector.Little-Endian</td>
<td>V.LE</td>
<td>Little-Endian support for Vector storage operations.</td>
</tr>
<tr>
<td>Vector.Crypto</td>
<td>V.GBB</td>
<td>Cryptographic instruction set</td>
</tr>
<tr>
<td>Vector.RAID</td>
<td>V.XOR</td>
<td>Vector Permute-XOR instruction</td>
</tr>
<tr>
<td>Wait</td>
<td>WT</td>
<td>wait instruction; see Book II</td>
</tr>
<tr>
<td>64-Bit</td>
<td>64</td>
<td>Required for 64-bit implementations; not defined for 32-bit impl's</td>
</tr>
</tbody>
</table>

¹ Because of overlapping opcode usage, SPE is mutually exclusive with Vector and with Legacy Integer Multiply-Accumulate, and Legacy Integer Multiply-Accumulate is mutually exclusive with Vector.

² The SPE-dependent Floating-Point categories are collectively referred to as SPE.Embedded Float_* or SP.*.

Figure 1. Category Listing (Sheet 2 of 2)

An instruction in a category that is not supported by the implementation is treated as an illegal instruction or an unimplemented instruction on that implementation (see Section 1.7.2).

For an instruction that is supported by the implementation with field values that are defined by the architecture, the field values defined as part of a category that is not supported by the implementation are treated as reserved values on that implementation (see Section 1.3.3 and Section 1.8.2).

Bits in a register that are in a category that is not supported by the implementation are treated as reserved.

1.3.5.1 Phased-In/Phased-Out

There are two special categories, Phased-In and Phased-Out, as well as two additional variations of Phased-In as defined below. Abbreviations, if applicable, are shown in parentheses.

Phased-In
These are facilities and instructions that, in the next version of the architecture, will be required as part of the category they are dependent on.

Servers do not implement a facility or instruction in this category. Servers that comply with earlier versions of this architecture may have optionally implemented facilities or instructions that were category Phased-In.

Server, Embedded.Phased-In (S,E.PI)

These are facilities and instructions that are part of the Server environment and, in the next version of the architecture, will be required for the Embedded environment.

It is implementation-dependent whether Embedded processors implement a facility or instruction in this category.

Embedded, Server.Phased-In (E,S.PI)

These are facilities and instructions that are part of the Embedded environment and, in the next version of the architecture, will be required for the Server environment.

Servers do not implement a facility or instruction in this category.

Phased-Out

These are facilities and instructions that, in some future version of the architecture, will be dropped out of the architecture. System developers should develop a migration plan to eliminate use of them in new systems.

For Server platforms, Phased-Out facilities and instructions must be implemented if the facility or instruction is part of another category (including the Base category) that is supported by the Server platform.

1.3.5.3 Category Notation

Instructions and facilities are considered part of the Base category unless otherwise marked. If a section is marked with a specific category tag, all material in that section and its subsections are considered part of the category, unless otherwise marked. Overview sections may contain discussion of instructions and facilities from various categories without being explicitly marked.

An example of a category tag is: [Category: Server]. Alternatively, a shorthand notation of a category tag includes the category name in angled brackets “<>”, such as <E.HV>.

An example of a dependent category is: [Category: Server.Phased-In]

The shorthand <E> and <S> may also be used for Category: Embedded and Server respectively.

1.3.6 Environments

All implementations support one of the two defined environments, Server or Embedded. Environments refer to common subsets of instructions that are shared across many implementations. The Server environment describes implementations that support Category: Base and Server. The Embedded environment describes implementations that support Category: Base and Embedded.

Warning: Instructions and facilities being phased out of the architecture are likely to perform poorly on future implementations. New programs should not use them.

Facilities are categorized as Phased-In only in cases where there is a difference between the Server and Embedded environments.

1.3.5.2 Corequisite Category

A corequisite category is an additional category that is associated with an instruction or facility, and must be implemented if the instruction or facility is implemented.
1.4 Processor Overview

The basic classes of instructions are as follows:

- branch instructions (Chapter 2)
- GPR-based scalar fixed-point instructions (Chapter 3, Chapter 9, and Chapter 11)
- GPR-based vector fixed-point instructions (Chapter 8)
- GPR-based scalar and vector floating-point instructions (Chapter 10)
- FPR-based scalar floating-point instructions (Chapter 4)
- FPR-based scalar decimal floating-point instructions (Chapter 6)
- VR-based vector fixed-point and floating-point instructions (Chapter 5)
- VSR-based scalar and vector floating-point instructions (Chapter 7)

Scalar fixed-point instructions operate on byte, halfword, word, doubleword, and quadword (see Book III-S) operands, where each operand contained in a GPR. Vector fixed-point instructions operate on vectors of byte, halfword, and word operands, where each vector is contained in a GPR or VR. Scalar floating-point instructions operate on single-precision or double-precision floating-point operands, where each operand is contained in a GPR, FPR, or VSR. Vector floating-point instructions operate on vectors of single-precision and double-precision floating-point operands, where each vector is contained in a GPR, VR, or VSR.

The Power ISA uses instructions that are four bytes long and word-aligned (VLE has different instruction characteristics; see Book VLE). It provides for byte, halfword, word, doubleword, and quadword operand loads and stores between storage and a set of 32 General Purpose Registers (GPRs). It provides for word and doubleword operand loads and stores between storage and a set of 32 Floating-Point Registers (FPRs). It also provides for byte, halfword, word, and quadword operand loads and stores between storage and a set of 32 Vector Registers (VRs). It provides for doubleword and quadword operand loads and stores between storage and a set of 64 Vector-Scalar Registers (VSRs).

Signed integers are represented in two’s complement form.

There are no computational instructions that modify storage; instructions that reference storage may reformat the data (e.g. load halfword algebraic). To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location.

Figure 2 is a logical representation of instruction processing. Figure 3 shows the registers that are defined in Book I. (A few additional registers that are available to application programs are defined in other Books, and are not shown in the figure.)
**Figure 3. Registers that are defined in Book I**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>32</td>
<td>63</td>
</tr>
<tr>
<td>LR</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>CTR</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>GPR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XER</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>VRSAVE</td>
<td>32</td>
<td>63</td>
</tr>
<tr>
<td>SPRG4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPRG5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPRG6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPRG7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Category: Embedded:**

**Category: Floating-Point:**

**Category: Vector:**

**Category: Vector-Scalar Extension:**

**Category: SPE:**
1.5 Computation modes

1.5.1 Modes [Category: Server]  
Processors provide two execution modes, 64-bit mode and 32-bit mode. In both of these modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how Condition Register bits and XER bits are set, how the Link Register is set by Branch instructions in which LK=1, and how the Count Register is tested by Branch Conditional instructions. Nearly all instructions are available in both modes (the only exceptions are a few instructions that are defined in Book III-S). In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers, Link Register, Count Register, etc.) and produce a 64-bit result. However, in 32-bit mode the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage; see Section 1.10.3 for additional details.

--- Programming Note ---
Although instructions that set a 64-bit register affect all 64 bits in both 32-bit and 64-bit modes, operating systems often do not preserve the upper 32-bits of all registers across context switches done in 32-bit mode. For this reason, application programs operating in 32-bit mode should not assume that the upper 32 bits of the GPRs are preserved from instruction to instruction unless the operating system is known to preserve these bits.

1.5.2 Modes [Category: Embedded]  
64-bit processors provide 64-bit mode and 32-bit mode. The differences between the two modes are described below. 32-bit processors provide only 32-bit mode, and do so as described at the end of this section.

- In 64-bit mode, the processor behaves as described for 64-bit mode in the Server environment; see Section 1.5.1.
- In 32-bit mode, the processor behavior depends on whether the high-order 32 bits of GPRs are implemented in 32-bit mode, as follows.
  - If these bits are implemented in 32-bit mode, the processor behaves as described for 32-bit mode in the Server environment.
  - If these bits are not implemented in 32-bit mode, the processor behaves as described for 32-bit mode in the Server Environment except for the following.

32-bit processors provide only 32-bit mode, and provide it as described by the second sub-bullet of the 32-bit mode bullet above.

1.6 Instruction Formats  
All instructions are four bytes long and word-aligned (except for VLE instructions; see Book VLE). Thus, whenever instruction addresses are presented to the processor (as in Branch instructions) the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

Bits 0:5 always specify the opcode (OPCD, below). Many instructions also have an extended opcode (XO, below). The remaining bits of the instruction contain one or more fields as shown below for the different instruction formats.
The format diagrams given below show horizontally all valid combinations of instruction fields. The diagrams include instruction fields that are used only by instructions defined in Book II or in Book III.

**Split Field Notation**

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits that are used in permuted order. Such a field is called a split field. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. In all other places, the name of the field is capitalized and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.

### 1.6.1 I-FORM

![Figure 4. I instruction format](image)

### 1.6.2 B-FORM

![Figure 5. B instruction format](image)

### 1.6.3 SC-FORM

![Figure 6. SC instruction format](image)

### 1.6.4 D-FORM

![Figure 7. D instruction format](image)

### 1.6.5 DS-FORM

![Figure 8. DS instruction format](image)

### 1.6.6 DQ-FORM

![Figure 9. DQ instruction format](image)
1.6.7 X-FORM

| OPCD | RT | RA | / | XO | / |
| OPCD | RT | RA | RB | XO | / |
| OPCD | RT | RA | RB | XO | EH |
| OPCD | RT | RA | NB | XO | / |
| OPCD | RT | / | SR | / | XO | / |
| OPCD | RT | /// | RB | XO | 1 |
| OPCD | RT | /// | RB | XO | 1 |
| OPCD | RS | RA | RB | XO | Rc |
| OPCD | RS | RA | RB | XO | / |
| OPCD | RS | RA | RB | XO | / |
| OPCD | RS | RA | NB | XO | / |
| OPCD | RS | RA | SH | XO | Rc |
| OPCD | RS | RA | /// | XO | Rc |
| OPCD | RS | RA | /// | XO | / |
| OPCD | RS | /// | SR | /// | XO | / |
| OPCD | RS | /// | RB | XO | / |
| OPCD | RS | /// | RB | XO | / |
| OPCD | TH | RA | RB | XO | / |
| OPCD | BF | /// | L | RA | RB | XO | / |
| OPCD | BF | /// | FRA | FRB | XO | / |
| OPCD | BF | /// | BFA | /// | XO | / |
| OPCD | BF | /// | M | U | /// | XO | Rc |
| OPCD | BF | /// | /// | /// | XO | / |
| OPCD | TH | RA | RB | XO | / |
| OPCD | CT | /// | /// | /// | XO | / |
| OPCD | CT | /// | RA | RB | XO | / |
| OPCD | /// | L | RA | RB | XO | / |
| OPCD | /// | L | /// | RB | XO | / |
| OPCD | /// | L | /// | /// | XO | / |
| OPCD | /// | L | /// | E | /// | XO | / |
| OPCD | TO | RA | RB | XO | / |
| OPCD | FRT | RA | RB | XO | / |
| OPCD | FRT | FRA | FRB | XO | / |
| OPCD | FRTp | RA | RB | XO | / |
| OPCD | FRT | /// | FRB | XO | Rc |
| OPCD | FRT | /// | FRBp | XO | Rc |
| OPCD | FRT | /// | /// | XO | Rc |
| OPCD | FRTp | /// | FRB | XO | Rc |
| OPCD | FRTp | /// | FRBp | XO | Rc |
| OPCD | FRTp | FRA | FRBp | XO | Rc |
| OPCD | FRTp | FRAp | FRBp | XO | Rc |
| OPCD | BF | /// | FRA | FRBp | XO | / |
| OPCD | BF | /// | FRAp | FRBp | XO | / |

Figure 10. X Instruction Format

1.6.8 XL-FORM

| OPCD | BT | BA | BB | XO | / |
| OPCD | BO | BI | /// | BH | XO | LK |
| OPCD | /// | S | XO | / |
| OPCD | BF | /// | BFA | /// | /// | XO | / |
| OPCD | /// | /// | /// | XO | / |
| OPCD | OC | XO | / |

Figure 11. XL instruction format

1.6.9 XFX-FORM

| OPCD | RT | spr | XO | / |
| OPCD | RT | tbr | XO | / |
| OPCD | RT | 0 | /// | XO | / |
| OPCD | RT | 1 | FXM | /// | XO | / |
| OPCD | RT | dcr | XO | / |
| OPCD | RT | pmrn | XO | / |
| OPCD | DUI | DUIIS | XO | / |
| OPCD | RS | 0 | FXM | /// | XO | / |
| OPCD | RS | 1 | FXM | /// | XO | / |
| OPCD | RS | spr | XO | / |
| OPCD | RS | dcr | XO | / |
| OPCD | RS | pmrn | XO | / |

Figure 12. XFX instruction format
1.6.10 XFL-FORM

| OPCD | LI | FLM | W | FRB | XO | Rc |

Figure 13. XFL instruction format

1.6.11 XX1-FORM

| OPCD | T | RA | RB | XO | T |
| OPCD | S | RA | RB | XO | S |

Figure 14. XX1 Instruction Format

1.6.12 XX2-FORM

| OPCD | T | /// | B | XO | B/T |
| OPCD | T | /// | UIM | B | XO | B/T |
| OPCD | BF | /// | B | XO | B/T |

Figure 15. XX2 Instruction Format

1.6.13 XX3-FORM

| OPCD | T | A | B | XO | W/B/T |
| OPCD | T | A | B | Sh | XO | W/B/T |
| OPCD | BF | /// | A | B | XO | W/B/T |
| OPCD | T | A | B | XO | W/BITX |
| OPCD | T | A | B | XO | DM | XO | W/BITX |

Figure 16. XX3 Instruction Format

1.6.14 XX4-FORM

| OPCD | T | A | B | C | XO | W/BITX |

Figure 17. XX4-Form Instruction Format

1.6.15 XS-FORM

| OPCD | RS | RA | sh | XO | sh | Rc |

Figure 18. XS instruction format

1.6.16 XO-FORM

| OPCD | RT | RA | RB | OE | XO | Rc |
| OPCD | RT | RA | RB | / | XO | Rc |
| OPCD | RT | RA | RB | / | XO | / |
| OPCD | RT | RA | /// | OE | XO | Rc |

Figure 19. XO instruction format

1.6.17 A-FORM

| OPCD | FRT | FRA | FRB | FRC | XO | Rc |
| OPCD | FRT | FRA | FRB | /// | XO | Rc |
| OPCD | FRT | FRA | /// | FRC | XO | Rc |
| OPCD | FRT | /// | FRB | /// | XO | Rc |

Figure 20. A instruction format

1.6.18 M-FORM

| OPCD | RS | RA | RB | MB | ME | Rc |
| OPCD | RS | RA | SH | MB | ME | Rc |

Figure 21. M instruction format

1.6.19 MD-FORM

| OPCD | RS | RA | sh | mb | XO | sh | Rc |
| OPCD | RS | RA | sh | me | XO | sh | Rc |

Figure 22. MD instruction format

1.6.20 MDS-FORM

| OPCD | RS | RA | RB | mb | XO | Rc |
| OPCD | RS | RA | RB | me | XO | Rc |

Figure 23. MDS instruction format

1.6.21 VA-FORM

| OPCD | VRT | VRA | VRB | VRC | XO |
| OPCD | VRT | VRA | VRB | /// | SHB | XO |

Figure 24. VA instruction format

1.6.22 VC-FORM

| OPCD | VRT | VRA | VRB | Rc | XO |

Figure 25. VC instruction format
### 1.6.23 VX-FORM

![Figure 26. VX instruction format](image1)

### 1.6.24 EVX-FORM

![Figure 27. EVX instruction format](image2)

### 1.6.25 EVS-FORM

![Figure 28. EVS instruction format](image3)

### 1.6.26 Z22-FORM

![Figure 29. Z22 instruction format](image4)

### 1.6.27 Z23-FORM

![Figure 30. Z23 instruction format](image5)

### 1.6.28 Instruction Fields

**A (6)**

Field used by the `tbegin` instruction to specify an implementation-specific function.

Field used by the `tend` instruction to specify the completion of the outer transaction and all nested transactions.

**AA (30)**

Absolute Address bit.

0 The immediate field represents an address relative to the current instruction address. For I-form branches the effective address of the branch target is the sum of the LI field sign-extended to 64 bits and the address of the branch instruction. For B-form branches the effective address of the branch target is the sum of the BD field sign-extended to 64 bits and the address of the branch instruction.

1 The immediate field represents an absolute address. For I-form branches the effective address of the branch target is the LI field sign-extended to 64 bits. For B-form branches the effective address of the branch target is the BD field sign-extended to 64 bits.

**AX (29) & A(11:15)**

Fields that are concatenated to specify a VSR to be used as a source.

**BA (11:15)**

Field used to specify a bit in the CR to be used as a source.

**BB (16:20)**

Field used to specify a bit in the CR to be used as a source.

**BC (21:25)**
Field used to specify a bit in the CR to be used as a source.

**BD (16:29)**
Immediate field used to specify a 14-bit signed two's complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

**BF (6:8)**
Field used to specify one of the CR fields or one of the FPSCR fields to be used as a target.

**BFA (11:13 or 29:31)**
Field used to specify one of the CR fields or one of the FPSCR fields to be used as a source.

**BH (19:20)**
Field used to specify a hint in the Branch Conditional to Link Register and Branch Conditional to Count Register instructions. The encoding is described in Section 2.5, “Branch Instructions”.

**BI (11:15)**
Field used to specify a bit in the CR to be tested by a Branch Conditional instruction.

**BO (6:10)**
Field used to specify options for the Branch Conditional instructions. The encoding is described in Section 2.5, “Branch Instructions”.

**BT (6:10)**
Field used to specify a bit in the CR or in the FPSCR to be used as a target.

**BX (30) & B(16:20)**
Fields that are concatenated to specify a VSR to be used as a source.

**CT (7:10)**
Field used in X-form instructions to specify a cache target (see Section 4.3.2 of Book II).

**CX (28) & C(21:25)**
Fields that are concatenated to specify a VSR to be used as a source.

**D (16:31)**
Immediate field used to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.

**DCM (16:21)**
Immediate field used as the Data Class Mask.

**DCR (11:20)**
Field used by the Move To/From Device Control Register instructions (see Book III-E).

**DGM (16:21)**
Immediate field used as the Data Group Mask.

**DM (24:25)**
Immediate field used by *xxpermdi* instruction as doubleword permute control.

**DQ (16:27)**
Immediate field used to specify a 12-bit signed two’s complement integer which is concatenated on the right with 0b0000 and sign-extended to 64 bits.

**DS (16:29)**
Immediate field used to specify a 14-bit signed two’s complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

**DUI (6:10)**
Field used by the *dnh* instruction (see Book III-E).

**DUIS (11:20)**
Field used by the *dnh* instruction (see Book III-E).

**E (16)**
Field used by the Write MSR External Enable instruction (see Book III-E).

**E (12:15)**
Field used to specify the access types ordered by an Elemental Memory Barrier type of *sync* instruction.

**EH (31)**
Field used to specify a hint in the Load and Reserve instructions. The meaning is described in Section 4.4.2, “Load and Reserve and Store Conditional Instructions”, in Book II.

**FLM (7:14)**
Field mask used to identify the FPSCR fields that are to be updated by the *mttsf* instruction.

**FRA (11:15)**
Field used to specify an FPR to be used as a source.

**FRAp (11:15)**
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

**FRB (16:20)**
Field used to specify an FPR to be used as a source.

**FRBp (16:20)**
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

**FRC (21:25)**
Field used to specify an FPR to be used as a source.

**FRS (6:10)**
Field used to specify an FPR to be used as a source.

FRSp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

FRT (6:10)
Field used to specify an FPR to be used as a target.

FRTp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a target.

FXM (12:19)
Field mask used to identify the CR fields that are to be written by the mtcfr and mtocfr instructions, or read by the mtocfr instruction.

IH (8:10)
Field used to specify a hint in the SLB Invalidate All instruction. The meaning is described in Section 5.9.3.1, "SLB Management Instructions", in Book III-S.

L (6)
Field used to specify whether the mtsfsf instruction updates the entire FPSCR.

L (10 or 15)
Field used to specify whether a fixed-point Compare instruction is to compare 64-bit numbers or 32-bit numbers.

Field used by the Data Cache Block Flush instruction (see Section 4.3.2 of Book II).

Field used by the Move To Machine State Register and TLB Invalidate Entry instructions (see Book III).

L (9:10)
Field used by the Data Cache Block Flush instruction (see Section 4.3.2 of Book II) and also by the Synchronize instruction (see Section 4.4.3 of Book II).

LEV (20:26)
Field used by the System Call instruction.

LI (6:29)
Immediate field used to specify a 24-bit signed two’s complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

LK (31)
LINK bit.
0 Do not set the Link Register.
1 Set the Link Register. The address of the instruction following the Branch instruction is placed into the Link Register.

MB (21:25) and ME (26:30)
Fields used in M-form instructions to specify a 64-bit mask consisting of 1-bits from bit MB+32 through bit ME+32 inclusive and 0-bits elsewhere, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 91.

MB (21:26)
Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 91.

ME (21:26)
Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 91.

MO (6:10)
Field used in X-form instructions to specify a subset of storage accesses.

NB (16:20)
Field used to specify the number of bytes to move in an immediate Move Assist instruction.

OC (6:20)
Field used by the Embedded Hypervisor Privilege instruction.

OPCD (0:5)
Primary opcode field.

OE (21)
Field used by XO-form instructions to enable setting OV and SO in the XER.

PMRN (11:20)
Field used to specify a Performance Monitor Register for the mfpmr and mtpmr instructions.

R (10)
Field used by the tbegin instruction to specify the start of a ROT.

R (15)
Immediate field that specifies whether the RMC is specifying the primary or secondary encoding.

RA (11:15)
Field used to specify a GPR to be used as a source or as a target.

RB (16:20)
Field used to specify a GPR to be used as a source.

Rc (21 OR 31)
RECORD bit.
0 Do not alter the Condition Register.
1 Set Condition Register Field 0, Field 1, or Field 6 as described in Section 2.3.1, “Condition Register” on page 30.

**RMC (21:22)**
Immediate field used for DFP rounding mode control.

**RS (6:10)**
Field used to specify a GPR to be used as a source.

**RSp (6:10)**
Field used to specify an even/odd pair of GPRs to be concatenated and used as a source.

**RT (6:10)**
Field used to specify a GPR to be used as a target.

**RTp (6:10)**
Field used to specify an even/odd pair of GPRs to be concatenated and used as a target.

**S (11 or 20)**
Immediate field that specifies signed versus unsigned conversion. Immediate field that specifies whether or not the rfebb instruction re-enables event-based branches.

**SH (16:20, or 16:20 and 30, or 16:21)**
Field used to specify a shift amount.

**SHB (22:25)**
Field used to specify a shift amount in bytes.

**SHW (24:25)**
Field used to specify a shift amount in words.

**SI (16:31 or 11:15)**
Immediate field used to specify a 16-bit signed integer.

**SIM (11:15)**
Immediate field used to specify a 5-bit signed integer.

**SP (11:12)**
Immediate field that specifies signed versus unsigned conversion.

**SPR (11:20)**
Field used to specify a Special Purpose Register for the mtsp and mfspr instructions.

**SR (12:15)**
Field used by the Segment Register Manipulation instructions (see Book III-S).

**SX (31) & S(6:10)**
Fields that are concatenated to specify a VSR to be used as a source.

Field used to specify the type of invalidation done by a TLB Invalidate Local instruction (see Book III-E).

**TBR (11:20)**
Field used by the Move From Time Base instruction (see Section 6.2.1 of Book II).

**TE (11:15)**
Immediate field that specifies a DFP exponent.

**TH (6:10)**
Field used by the data stream variant of the dcbt and dcbtsi instructions (see Section 4.3.2 of Book II).

**TO (6:10)**
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.11, “Fixed-Point Trap Instructions” on page 80.

**TX (31) & T (6:10)**
Fields that are concatenated to specify a VSR to be used as a target.

**U (16:19)**
Immediate field used as the data to be placed into a field in the FPSCR.

**UI (11:15, 16:20, or 16:31)**
Immediate field used to specify an unsigned integer.

**UIM (11:15, 12:15, 13:15, 14:15)**
Immediate field used to specify an unsigned integer.

**VRA (11:15)**
Field used to specify a VR to be used as a source.

**VRB (16:20)**
Field used to specify a VR to be used as a source.

**VRC (21:25)**
Field used to specify a VR to be used as a source.

**VRS (6:10)**
Field used to specify a VR to be used as a source.

**VRT (6:10)**
Field used to specify a VR to be used as a target.

**W (15)**
Field used by the mtfsi and mtfs instructions to specify the target word in the FPSCR.

**WC (9:10)**
Field used to specify the condition or conditions that cause instruction execution to resume after executing a wait [Category: Wait] instruction (see Section 4.4.4 of Book II).
1.7 Classes of Instructions

An instruction falls into exactly one of the following three classes:

- Defined
- Illegal
- Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or a reserved instruction, the instruction is illegal.

1.7.1 Defined Instruction Class

This class of instructions contains all the instructions defined in this document.

A defined instruction can have preferred and/or invalid forms, as described in Section 1.8.1, “Preferred Instruction Forms” and Section 1.8.2, “Invalid Instruction Forms”. Instructions that are part of a category that is not supported are treated as illegal instructions.

1.7.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix D of Book Appendices. Illegal instructions are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0s is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

1.7.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix E of Book Appendices.

Reserved instructions are allocated to specific purposes that are outside the scope of the Power ISA.

Any attempt to execute a reserved instruction will:

- perform the actions described by the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.
1.8 Forms of Defined Instructions

1.8.1 Preferred Instruction Forms

Some of the defined instructions have preferred forms. For such an instruction, the preferred form will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form. Instructions having preferred forms are:

- the Condition Register Logical instructions
- the Load/Store Multiple instructions
- the Load/Store String instructions
- the Or Immediate instruction (preferred form of no-op)
- the Move To Condition Register Fields instruction

1.8.2 Invalid Instruction Forms

Some of the defined instructions can be coded in a form that is invalid. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

In general, any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 1.3.3 on page 5); these invalid forms are not discussed further. All other invalid forms are identified in the instruction descriptions.

References to instructions elsewhere in this document assume the instruction form is not invalid, unless otherwise stated or obvious from context.

Assembler Note

Assemblers should report uses of invalid instruction forms as errors.

1.8.3 Reserved-no-op Instructions

[Category: Phased-In]

Reserved-no-op instructions include the following extended opcodes under primary opcode 31: 530, 562, 594, 626, 658, 690, 722, and 754.

Reserved-no-op instructions are provided in the architecture to anticipate the eventual adoption of performance hint instructions to the architecture. For these instructions, which cause no visible change to architected state, employing a reserved-no-op opcode will allow software to use this new capability on new implementations that support it while remaining compatible with existing implementations that may not support the new function.

When a reserved-no-op instruction is executed, no operation is performed.

Reserved-no-op instructions are not assigned instruction names or mnemonics. There are no individual descriptions of reserved-no-op instructions in this document.

1.9 Exceptions

There are two kinds of exception, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a “privileged” instruction (see Book III) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- an attempt to execute an instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to access a storage location that is unavailable (system instruction storage error handler or system data storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)
- the execution of a System Call instruction (system service program)
- the execution of a Trap instruction that traps (system trap handler)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (system floating-point enabled exception error handler)
- the execution of an auxiliary processor instruction that causes an auxiliary processor enabled exception to exist (system auxiliary processor enabled exception error handler)

The exceptions that can be caused by an asynchronous event are described in Book III.
The invocation of the system error handler is precise, except that the invocation of the auxiliary processor enabled exception error handler may be imprecise, and if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 125), then the invocation of the system floating-point enabled exception error handler may also be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in Book III.

### 1.10 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a Storage Access or Branch instruction (or certain other instructions described in Book II and Book III), or when it fetches the next sequential instruction.

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

The byte ordering (Big-Endian or Little-Endian) for a storage access is specified by the operating system. In the Embedded environment this ordering is a page attribute (see Book II) and is specified independently for each virtual page, while in the Server environment it is specified by the two mode bits, LE and SLE (see Book III-S), and applies to all storage.

#### 1.10.1 Storage Operands

A storage operand may be a byte, a halfword, a word, a doubleword, or a quadword, or, for the Load/Store Multiple and Move Assist instructions, a sequence of bytes (Move Assist) or words (Load/Store Multiple). For example, the storage operand of a Load Floating-Point Double Pair instruction is a quadword, not two double-words. The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). An instruction for which the storage operand is a byte is said to cause a byte access, and similarly for halfword, word, doubleword, and quadword.

Operand length is implicit for each instruction.

The storage operand of a single-register Storage Access instruction or quadword Load or Store instruction has a “natural” alignment boundary equal to the operand length. In other words, the “natural” address of such an operand is an integral multiple of the operand length. Such an operand is said to be aligned if it is aligned at its natural boundary; otherwise it is said to be unaligned. See the following table.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Length</th>
<th>Addr60:63 if aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>xxxxx</td>
</tr>
<tr>
<td>Halfword</td>
<td>2 bytes</td>
<td>xxx0</td>
</tr>
<tr>
<td>Word</td>
<td>4 bytes</td>
<td>xx00</td>
</tr>
<tr>
<td>Doubleword</td>
<td>8 bytes</td>
<td>x000</td>
</tr>
<tr>
<td>Quadword</td>
<td>16 bytes</td>
<td>0000</td>
</tr>
</tbody>
</table>

Note: An “x” in an address bit position indicates that the bit can be 0 or 1 independent of the contents of other bits in the address.

The concept of alignment is also applied more generally, to any datum in storage. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. For single-register Storage Access instructions and quadword Load and Store instructions, the best performance is obtained when storage operands are aligned. Additional effects of data placement on performance are described in Chapter 2 of Book II.

When a storage operand of length N bytes starting at effective address EA is copied between storage and a register that is R bytes long (i.e., the register contains bytes numbered from 0, most significant, through R-1, least significant), the bytes of the operand are placed into the register or into storage in a manner that depends on the byte ordering for the storage access as shown in Figure 31, unless otherwise specified in the instruction description.

<table>
<thead>
<tr>
<th>Big-Endian Byte Ordering</th>
<th>Little-Endian Byte Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>for i=0 to N-1:</td>
<td>for i=0 to N-1:</td>
</tr>
<tr>
<td>RT<a href="R-N">RS</a>+i&lt;-MEM(EA+i,1)</td>
<td>MEM(EA+i,1)&lt;-(RS)[R-N]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>for i=0 to N-1:</td>
<td>for i=0 to N-1:</td>
</tr>
<tr>
<td>RT<a href="R-1">RS</a>+i&lt;-MEM(EA+i,1)</td>
<td>MEM(EA+i,1)&lt;-(RS)[R-1]</td>
</tr>
</tbody>
</table>

Notes:
1. In this table, subscripts refer to bytes in a register rather than to bits as defined in Section 1.3.2.
2. This table does not apply to the **lvebx**, **lvehx**, **lvewx, stvebx, stvehx**, and **stvewx** instructions.

**Figure 31. Storage operands and byte ordering**

Figure 32 shows an example of a C language structure **s** containing an assortment of scalars and one character string. The value assumed to be in each structure element is shown in hex in the C comments; these values are used below to show how the bytes making up each structure element are mapped into storage. It is assumed that structure s is compiled for
32-bit mode or for a 32-bit implementation. (This affects the length of the pointer to c.)

C structure mapping rules permit the use of padding (skipped bytes) in order to align the scalars on desirable boundaries. Figures 33 and 34 show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between a and b, one byte between d and e, and two bytes between e and f. The same amount of padding is present for both Big-Endian and Little-Endian mappings.

The Big-Endian mapping of structure s is shown in Figure 33. Addresses are shown in hex at the left of each doubleword, and in small figures below each byte. The contents of each byte, as indicated in the C example in Figure 32, are shown in hex (as characters for the elements of the string).

The Little-Endian mapping of structure s is shown in Figure 34. Doublewords are shown laid out from right to left, which is the common way of showing storage maps for processors that implement only Little-Endian byte ordering.

The Big-Endian mapping of program p is shown in Figure 37 (assuming the program starts at address 0).

struct {
  int a; /* 0x1112_1314 */
  double b; /* 0x2122_2324_2526_2728 */
  char * c; /* 0x3132_3334 */
  char d[7]; /* 'A', 'B', 'C', 'D', 'E', 'F', 'G' */
  short e; /* 0x5152 */
  int f; /* 0x6162_6364 */
} s;

Figure 32. C structure ‘s’, showing values of elements

```
<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>08</th>
<th>10</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>00</td>
<td>21</td>
<td>18</td>
<td>61</td>
</tr>
<tr>
<td>01</td>
<td>12</td>
<td>01</td>
<td>22</td>
<td>19</td>
<td>62</td>
</tr>
<tr>
<td>02</td>
<td>13</td>
<td>02</td>
<td>23</td>
<td>1A</td>
<td>63</td>
</tr>
<tr>
<td>03</td>
<td>14</td>
<td>03</td>
<td>24</td>
<td>1B</td>
<td>64</td>
</tr>
<tr>
<td>04</td>
<td></td>
<td>04</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td></td>
<td>05</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td></td>
<td>06</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
<td>07</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 33. Big-Endian mapping of structure ‘s’

```
<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>08</th>
<th>10</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>51</td>
<td>00</td>
<td>31</td>
<td>18</td>
<td>61</td>
</tr>
<tr>
<td>01</td>
<td>52</td>
<td>01</td>
<td>32</td>
<td>19</td>
<td>62</td>
</tr>
<tr>
<td>02</td>
<td></td>
<td>02</td>
<td>33</td>
<td>1A</td>
<td>63</td>
</tr>
<tr>
<td>03</td>
<td></td>
<td>03</td>
<td>34</td>
<td>1B</td>
<td>64</td>
</tr>
<tr>
<td>04</td>
<td></td>
<td>04</td>
<td></td>
<td>1C</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td></td>
<td>05</td>
<td></td>
<td>1D</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td></td>
<td>06</td>
<td></td>
<td>1E</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
<td>07</td>
<td></td>
<td>1F</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 34. Little-Endian mapping of structure ‘s’
Figure 37. Big-Endian mapping of program ‘p’

The Little-Endian mapping of program p is shown in Figure 38.

Figure 38. Little-Endian mapping of program ‘p’
Programming Note

The terms Big-Endian and Little-Endian come from Part I, Chapter 4, of Jonathan Swift's Gulliver's Travels. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

... our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of Lilliput and Blefuscu. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of Blefuscu; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the Big-Endians have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of Blefuscu did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet Lustrog, in the fifty-fourth Chapter of the Brundrecal, (which is their Alcoran.) This, however, is thought to be a mere Strain upon the text: For the Words are these; That all true Believers shall break their Eggs at the convenient End: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the Big-Endian Exiles have found so much Credit in the Emperor of Blefuscu's Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

1.10.3 Effective Address Calculation

An effective address is computed by the processor when executing a Storage Access or Branch instruction (or certain other instructions described in Book II, Book III, and Book VLE) when fetching the next sequential instruction, or when invoking a system error handler. The following provides an overview of this process. More detail is provided in the individual instruction descriptions.

Effective address calculations, for both data and instruction accesses, use 64-bit two's complement addition. All 64 bits of each address component participate in the calculation regardless of mode (32-bit or 64-bit). In this computation one operand is an address (which is by definition an unsigned number) and the second is a signed offset. Carries out of the most significant bit are ignored.

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, 2^{64} - 1, to address 0, except that if the current instruction is at effective address 2^{64} - 4 the effective address of the next sequential instruction is undefined.

In 32-bit mode, the low-order 32 bits of the 64-bit result, preceded by 32 0 bits, comprise the 64-bit effective address for the purpose of addressing storage. When an effective address is placed into a register by an instruction or event, the value placed into the high-order 32 bits of the register differs between the Server environment and the Embedded environment.

- Server environment, and Embedded Environment when the high-order 32 bits of GPRs are implemented:
  - Load with Update and Store with Update instructions set the high-order 32 bits of register RA to the high-order 32 bits of the 64-bit result.
- In all other cases (e.g., the Link Register when set by Branch instructions having \( LK=1 \), Special Purpose Registers when set to an effective address by invocation of a system error handler) the high-order 32 bits of the register are set to 0s except as described in the last sentence of this paragraph.

- Embedded environment when the high-order 32 bits of GPRs are not implemented for the following cases:
  The high-order 32 bits of the register are set to an undefined value except for the Initialize Next Instruction register [Category: Embedded.Multi-threading] (see Section 1.5.2 and Book III), and for the following case. For a register that is loaded with an effective address by the invocation of a system error handler, the high-order 32 bits of the register are set to 0s if the computation mode is 64-bit after the system error is invoked. The 64-bit current instruction address is not affected by a change from 32-bit mode to 64-bit mode, but is affected by a change from 64-bit mode to 32-bit mode. In the latter case, the high-order 32 bits are set to 0. The same is true for the 64-bit next instruction address, except as described in the last item of the list below.

As used to address storage, the effective address arithmetic appears to wrap around from the maximum address, \( 2^{32} - 1 \), to address 0, except that if the current instruction is at effective address \( 2^{32} - 4 \) the effective address of the next sequential instruction is undefined.

RA is a field in the instruction which specifies an address component in the computation of an effective address. A zero in the RA field indicates the absence of the corresponding address component. A value of zero is substituted for the absent component of the effective address computation. This substitution is shown in the instruction descriptions as (RA|0).

Effective addresses are computed as follows. In the descriptions below, it should be understood that “the contents of a GPR” refers to the entire 64-bit contents, independent of mode, but that in 32-bit mode only bits 32:63 of the 64-bit result of the computation are used to address storage.

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for `lswi` and `stswi`) are added to the contents of the GPR designated by RA or to zero if RA=0 or RA is not used in forming the EA.

- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- With I-form Branch instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.

- With B-form Branch instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.

- With XL-form Branch instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the target instruction.

- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction, except that if the current instruction is at the maximum instruction effective address for the mode (\( 2^{64} - 4 \) in 64-bit mode, \( 2^{32} - 4 \) in 32-bit mode) the effective address of the next sequential instruction is undefined. (There is one other exception to this rule; this exception involves changing between 32-bit mode and 64-bit mode and is described in Section 6.3.2 of Book III-E and Section 6.3.2 of Book III-E.)

If the size of the operand of a storage access instruction is more than one byte, the effective address for each byte after the first is computed by adding 1 to the effective address of the preceding byte.
Chapter 2. Branch Facility

2.1 Branch Facility Overview

This chapter describes the registers and instructions that make up the Branch Facility.

2.2 Instruction Execution Order

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- **Branch** instructions for which the branch is taken cause execution to continue at the target address specified by the Branch instruction.
- **Trap** instructions for which the trap conditions are satisfied, and **System Call** instructions, cause the appropriate system handler to be invoked.
- Transaction failure will eventually cause the transaction’s failure handler, implied by the `tbegin` instruction, to be invoked. See the programming note following the `tbegin` description in Section 5.5 of Book II.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.9, “Exceptions” on page 22.
- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.

The model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction is called the “sequential execution model.” In general, the processor obeys the sequential execution model. For the instructions and facilities defined in this Book, the only exceptions to this rule are the following.

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 4.4). The instruction that causes the exception need not complete before the next instruction begins execution, with respect to setting exception bits and (if the exception is enabled) invoking the system error handler.

- A Store instruction modifies one or more bytes in an area of storage that contains instructions that will subsequently be executed. Before an instruction in that area of storage is executed, software synchronization is required to ensure that the instructions executed are consistent with the results produced by the Store instruction.

---

**Programming Note**

This software synchronization will generally be provided by system library programs (see Section 1.9 of Book II). Application programs should call the appropriate system library program before attempting to execute modified instructions.
2.3 Branch Facility Registers

2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).

Figure 39. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (mtcrf, mtocrf).
- A specified field of the CR can be set by a move to the CR from another CR field (mcrf), from XER32:35 (mcxr), or from the FPSCR (mcrfs).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 1 can be set as the implicit result of a decimal floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a Compare instruction or of a tcheck instruction (see Book II).

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which Rc=1, and for addic, andi, and andis., the first three bits of CR Field 0 (bits 32:34 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 35 of the Condition Register) is copied from the SO field of the XER. “Result” here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode.

```lisp
if (64-bit mode)
    then M = 0
else M = 32
if (target_register)M,63 < 0 then c ← 0b100
else if (target_register)M,63 > 0 then c ← 0b010
else
    c ← 0b001
CR0 ← c || XERSO
```

If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

The bits of CR Field 0 are interpreted as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Negative</strong> (LT)</td>
</tr>
<tr>
<td></td>
<td>The result is negative.</td>
</tr>
<tr>
<td>1</td>
<td><strong>Positive</strong> (GT)</td>
</tr>
<tr>
<td></td>
<td>The result is positive.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Zero</strong> (EQ)</td>
</tr>
<tr>
<td></td>
<td>The result is zero.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Summary Overflow</strong> (SO)</td>
</tr>
<tr>
<td></td>
<td>This is a copy of the contents of XER_SO at the completion of the instruction.</td>
</tr>
</tbody>
</table>

With the exception of tcheck, the Transactional Memory instructions set CR0[2] indicating the state of the facility prior to instruction execution, or transaction failure. A complete description of the meaning of these bits is given in the instruction descriptions in Section 5.5 of Book II. These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>CR0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0</td>
<td>Transaction state of Non-transactional prior to instruction</td>
</tr>
<tr>
<td>010 0</td>
<td>Transaction state of Transactional prior to instruction</td>
</tr>
<tr>
<td>001 0</td>
<td>Transaction state of Suspended prior to instruction</td>
</tr>
<tr>
<td>101 0</td>
<td>Transaction failure</td>
</tr>
</tbody>
</table>

The tcheck instruction similarly sets bits 1 and 2 of CR field BF to indicate the transaction state, and additionally sets bit 0 to TDOOMED, as defined in Section 5.5 of Book II.

<table>
<thead>
<tr>
<th>CR field BF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDOOMED 0 0</td>
<td>Transaction state of Non-transactional prior to instruction</td>
</tr>
<tr>
<td>TDOOMED 1 0</td>
<td>Transaction state of Transactional prior to instruction</td>
</tr>
<tr>
<td>TDOOMED 0 1</td>
<td>Transaction state of Suspended prior to instruction</td>
</tr>
</tbody>
</table>

**Programming Note**

Setting of bit 3 of the specified CR field to zero by tcheck and of field CR0[3] to zero by other TM instructions is intended to preserve these bits for future function. Software should not depend on the bits being zero.

The stbcx., sthcx., stwcx., stdcx., and stqcx. instructions (see Section 4.4.2, “Load and Reserve and Store Operations”...
For all floating-point instructions in which Rc=1, CR Field 0.

Conditional Instructions”, in Book II) also set CR Field 0.

For all floating-point instructions in which Rc=1, CR Field 1 (bits 36:39 of the Condition Register) is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register. This occurs regardless of whether any exceptions are enabled, and regardless of whether the writing of the result is suppressed (see Section 4.4, “Floating-Point Exceptions” on page 124). These bits are interpreted as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
<tr>
<td>33</td>
<td>Floating-Point Enabled Exception Summary (FEX)</td>
</tr>
<tr>
<td>34</td>
<td>Floating-Point Invalid Operation Exception Summary (VX)</td>
</tr>
<tr>
<td>35</td>
<td>Floating-Point Overflow Exception (OX)</td>
</tr>
</tbody>
</table>

For Compare instructions, a specified CR field is set to reflect the result of the comparison. The bits of the specified CR field are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 3.3.10, “Fixed-Point Compare Instructions” on page 78, Section 4.6.8, “Floating-Point Compare Instructions” on page 159, and Section 8.3.9, “SPE Instruction Set” on page 594.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less Than, Floating-Point Less Than (LT, FL)</td>
</tr>
<tr>
<td>1</td>
<td>Greater Than, Floating-Point Greater Than (GT, FG)</td>
</tr>
<tr>
<td>2</td>
<td>Equal, Floating-Point Equal (EQ, FE)</td>
</tr>
</tbody>
</table>

For fixed-point Compare instructions, this is a copy of the contents of XERSO at the completion of the instruction. For floating-point Compare instructions, one or both of (FRA) and (FRB) is a NaN.

The Vector Integer Compare instructions (see Section 5.9.2, “Vector Integer Compare Instructions”) compare two Vector Registers element by element, interpreting the elements as unsigned or signed integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and 0s if the relation being tested is false.

If Rc=1, CR Field 6 is set to reflect the result of the comparison, as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The relation is true for all element pairs (i.e., VRT is set to all 1s).</td>
</tr>
<tr>
<td>1</td>
<td>The relation is false for all element pairs (i.e., VRT is set to all 0s).</td>
</tr>
</tbody>
</table>

The Vector Floating-Point Compare instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the Vector Compare Bounds Floating-Point instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the Vector Integer Compare instructions.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The relation is true for all element pairs (i.e., VRT is set to all 1s).</td>
</tr>
<tr>
<td>1</td>
<td>The relation is false for all element pairs (i.e., VRT is set to all 0s).</td>
</tr>
</tbody>
</table>

The Vector Compare Bounds Floating-Point instruction on page 251 sets CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be “within the bounds” specified by a single-precision floating-point value y if -y ≤ x ≤ y.
2. Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.

2.3.2 Link Register

The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the Branch Conditional to Link Register instruction, and it holds the return address after Branch instructions for which \( LK=1 \).

<table>
<thead>
<tr>
<th>LR</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 40. Link Register

2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is \(-1\) afterward. The Count Register can also be used to provide the branch target address for the Branch Conditional to Count Register instruction.

<table>
<thead>
<tr>
<th>CTR</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 41. Count Register

2.3.4 Target Address Register

The Target Address Register (TAR) is a 64-bit register. It can be used to provide bits 0:61 of the branch target address for the Branch Conditional to Branch Target Address Register instruction. Bits 62:63 are ignored by the hardware but can be set and reset by software.

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 42. Target Address Register

Programming Note

The TAR is reserved for system software.

2.4 Branch History Rolling Buffer [Category: Server]

The Branch History Rolling Buffer (BHRB) is a buffer containing an implementation-dependent number of entries, referred to as BHRB Entries (BHRBEs), that contain information related to branches that have been taken. Entries are numbered from 0 through \( n \), where \( n \) is implementation dependent. Entry 0 is the most-recently written entry.

The system program typically controls the availability of the BHRB as well as the number of entries that it contains. If the BHRB is accessed when it is unavailable, the Facility Unavailable error handler is invoked.

Various events or actions by the system program may result in the BHRB occasionally being cleared. If BHRB entries are read after this has occurred, 0s will be returned. See the description of the \texttt{mfbhrbe} instruction for additional information.

The BHRB is typically used in conjunction with Performance Monitor event-based branches. (See Chapter 7 of Book II.) When used in conjunction with this facility, BESC\textsubscript{RME} is set to 1 to enable Performance Monitor event-based branches, and Performance Monitor alerts are enabled to enable the writing of BHRB entries. (See Section 9.4.4 of Book III-S.) When a Performance Monitor alert occurs, Performance Monitor alerts are disabled, BHRB entries are no longer written, and an event-based branch occurs. (See Chapter 9 of Book III-S for additional information.) The event-based branch handler can then access the contents of the BHRB for analysis.

When the BHRB is written by hardware, only those Branch instructions that meet the filtering criterion are written. The filtering criterion is set by the system program. (See Section 9.4.7 of Book III-S.)

The following paragraphs describe the entries written into the BHRB for various types of Branch instructions for which the branch was taken. In some circumstances, however, the hardware may be unable to make the entry even though the following paragraphs require it. In such cases, the hardware sets the EA field to 0, and indicates any missed entries using the T and P fields. (See Section 2.4.1.)

When an I-form or B-form Branch instruction is entered into the BHRB, bits 0:61 of the branch instruction are written into the next available entry, except that the entry may or may not be written if the instruction following the Branch instruction is not another Branch instruction, and target address of the Branch instruction does not exceed its effective address by more than 8.

When a XL-form Branch instruction is entered into the BHRB, bits 0:61 of the effective address of the Branch instruction are placed into the next available entry, except that the entry may or may not be written if the instruction following the Branch instruction is another Branch instruction, and target address of the Branch instruction does not exceed its effective address by more than 8.

When a XL-form Branch instruction is entered into the BHRB, bits 0:61 of the branch instruction are written into the next available entry if allowed by the filtering mode; subsequently, bits 0:61 of the effective address of the Branch target is written into the following entry. The BHRB is read by means of the \texttt{mfbhrbe} instruction.

BHRB entries are written as described above without regard to transactional state and are not removed due to transaction failures.
2.4.1 Branch History Rolling Buffer Entry Format

Branch History Rolling Buffer Data Entries (BHRBEs) have the following format.

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>T</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 43. Branch History Rolling Buffer Data

When this field is set to a non-zero value, it contains bits 0:61 of the effective address of the instruction indicated by the T field; otherwise this field indicates that the entry is a marker with the meaning specified by the T and P fields.

When the EA field contains a non-zero value, bits 62:63 have the following meanings.

62  **Target Address (T)**

- 0 The EA field contains bits 0:61 of the effective address of a Branch instruction for which the branch was taken.
- 1 The EA field contains bits 0:61 of the branch target effective address corresponding to an XL-form Branch instruction.

63  **Prediction (P)**

When T=0, this field has the following meaning.

- 0 The outcome of the Branch instruction was correctly predicted.
- 1 The outcome of the Branch instruction was mispredicted.

When T=1, this field has the following meaning.

- 0 The Branch instruction was predicted to be taken and the target address was predicted correctly, or the target address was not predicted because the Branch instruction was predicted to be not taken.
- 1 The target address was mispredicted.

When the EA field contains a zero value, either the entry is a marker or is an entry for a Branch instruction for which the instruction address or target address is 0.

It is expected that programs will not contain Branch instructions with instruction or target addresses equal to 0. If such instructions exist, however, markers become unusable because BHRB entries for various markers cannot be distinguished from entries for Branch instructions with instruction or target addresses equal to 0.

For Branch instructions with zero instruction or target addresses, the EA field is set to 0 and bits 62:63 are specified above. For Branch instructions with non-zero effective and target addresses, the EA field is set to 0 and bits 62:63 specify the type of marker as described below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>This entry is either not implemented or has been cleared, and there are no valid entries beyond the current entry.</td>
</tr>
<tr>
<td>01</td>
<td>A Branch instruction for which the branch was taken was executed, but the hardware was unable to enter its effective address and, for XL-Form Branch instructions, its target effective address.</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>The previous entry contains bits 0:61 of the effective address of an XL-form Branch instruction for which the branch was taken, and the filtering mode required bits 0:61 of the current entry to indicate the effective address of the branch target, but the hardware was unable to enter the effective address of the branch target.</td>
</tr>
</tbody>
</table>
2.5 Branch Instructions

The sequence of instruction execution can be changed by the Branch instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are ignored by the processor in performing the branch.

The Branch instructions compute the effective address (EA) of the target in one of the following five ways, as described in Section 1.10.3, “Effective Address Calculation” on page 26.

1. Adding a displacement to the address of the Branch instruction (Branch or Branch Conditional with AA=0).
2. Specifying an absolute address (Branch or Branch Conditional with AA=1).
3. Using the address contained in the Link Register (Branch Conditional to Link Register).
4. Using the address contained in the Count Register (Branch Conditional to Count Register).
5. Using the address contained in the Target Address Register (Branch Conditional to Target Address Register).

In all five cases, in 32-bit mode the final step in the address computation is setting the high-order 32 bits of the target address to 0.

For the first two methods, the target addresses can be computed sufficiently ahead of the Branch instruction that instructions can be prefetched along the target path. For the third through fifth methods, prefetching instructions along the target path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the Branch instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the Branch instruction is placed into the Link Register after the branch target address has been computed; this is done regardless of whether the branch is taken.

For Branch Conditional instructions, the BO field specifies the conditions under which the branch is taken, as shown in Figure 44. In the figure, M=0 in 64-bit mode and M=32 in 32-bit mode.

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000z</td>
<td>Decrement the CTR, then branch if the decremented CTR_M:63≠0 and CR_Bi=0</td>
</tr>
<tr>
<td>0001z</td>
<td>Decrement the CTR, then branch if the decremented CTR_M:63=0 and CR_Bi=0</td>
</tr>
<tr>
<td>001at</td>
<td>Branch if CR_Bi=0</td>
</tr>
<tr>
<td>0100z</td>
<td>Decrement the CTR, then branch if the decremented CTR_M:63≠0 and CR_Bi=1</td>
</tr>
<tr>
<td>0101z</td>
<td>Decrement the CTR, then branch if the decremented CTR_M:63=0 and CR_Bi=1</td>
</tr>
<tr>
<td>011at</td>
<td>Branch if CR_Bi=1</td>
</tr>
<tr>
<td>1a00t</td>
<td>Decrement the CTR, then branch if the decremented CTR_M:63=0</td>
</tr>
<tr>
<td>1a01t</td>
<td>Decrement the CTR, then branch if the decremented CTR_M:63=0</td>
</tr>
<tr>
<td>1z1zz</td>
<td>Branch always</td>
</tr>
</tbody>
</table>

**Figure 44. BO field encodings**

The “a” and “t” bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken, as shown in Figure 45.

<table>
<thead>
<tr>
<th>at</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No hint is given</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>The branch is very likely not to be taken</td>
</tr>
<tr>
<td>11</td>
<td>The branch is very likely to be taken</td>
</tr>
</tbody>
</table>

**Figure 45. “at” bit encodings**

**Programming Note**

Many implementations have dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate, and is likely to be overridden by any hint provided by the “at” bits, the “at” bits should be set to 0b00 unless the static prediction implied by at=0b10 or at=0b11 is highly likely to be correct.

For Branch Conditional to Link Register and Branch Conditional to Count Register instructions, the BH field...
provides a hint about the use of the instruction, as shown in Figure 46.

<table>
<thead>
<tr>
<th>BH</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>bhcl[l]: The instruction is a subroutine return</td>
</tr>
<tr>
<td></td>
<td>bcctr[l] and bcctar[l]: The instruction is not a subroutine return;</td>
</tr>
<tr>
<td></td>
<td>the target address is likely to be the same as the target address</td>
</tr>
<tr>
<td></td>
<td>used the preceding time the branch was taken</td>
</tr>
<tr>
<td>01</td>
<td>bhcl[l]: The instruction is not a subroutine return; the target</td>
</tr>
<tr>
<td></td>
<td>address is likely to be the same as the target address used the</td>
</tr>
<tr>
<td></td>
<td>preceding time the branch was taken</td>
</tr>
<tr>
<td></td>
<td>bcctr[l] and bcctar[l]: Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>bhcl[l], bcctr[l], and bcctar[l]: The target address is not</td>
</tr>
<tr>
<td></td>
<td>predictable</td>
</tr>
</tbody>
</table>

**Figure 46. BH field encodings**

---

**Programming Note**

The hint provided by the BH field is independent of the hint provided by the “at” bits (e.g., the BH field provides no indication of whether the branch is likely to be taken).

**Extended mnemonics for branches**

Many extended mnemonics are provided so that Branch Conditional instructions can be coded with portions of the BO and BI fields as part of the mnemonic rather than as part of a numeric operand. Some of these are shown as examples with the Branch instructions. See Appendix E for additional extended mnemonics.

---

**Programming Note**

The hints provided by the “at” bits and by the BH field do not affect the results of executing the instruction.

The “z” bits should be set to 0, because they may be assigned a meaning in some future version of the architecture.
Many implementations have dynamic mechanisms for predicting the target addresses of `bclr`[l] and `bcctr`[l] instructions. These mechanisms may cache return addresses (i.e., Link Register values set by `Branch` instructions for which `LK`=1 and for which the branch was taken, other than the special form shown in the first example below) and recently used branch target addresses. To obtain the best performance across the widest range of implementations, the programmer should obey the following rules.

- **Use `Branch` instructions for which `LK`=1 only as subroutine calls (including function calls, etc.), or in the special form shown in the first example below.**

- **Pair each subroutine call (i.e., each `Branch` instruction for which `LK`=1 and the branch is taken, other than the special form shown in the first example below) with a `bclr` instruction that returns from the subroutine and has BH=0b00.**

- **Do not use `bclrl` as a subroutine call. (Some implementations access the return address cache at most once per instruction; such implementations are likely to treat `bclrl` as a subroutine return, and not as a subroutine call.)**

- **For `bclr`[l] and `bcctr`[l], use the appropriate value in the BH field.**

The following are examples of programming conventions that obey these rules. In the examples, BH is assumed to contain 0b00 unless otherwise stated. In addition, the "at" bits are assumed to be coded appropriately.

Let A, B, and Glue be specific programs.

- **Obtaining the address of the next instruction:**
  Use the following form of `Branch and Link`.
  
  \[
  \text{bcl} \quad 20,31,\$+4
  \]

- **Loop counts:**
  Keep them in the Count Register, and use a `bc` instruction (LK=0) to decrement the count and to branch back to the beginning of the loop if the decremented count is nonzero.

- **Computed goto’s, case statements, etc.:**
  Use the Count Register to hold the address to branch to, and use a `bcctr` instruction (LK=0, and BH=0b11 if appropriate) to branch to the selected address.

- **Direct subroutine linkage:**
  Here A calls B and B returns to A. The two branches should be as follows.
  - A calls B: use a `bl` or `bcl` instruction (LK=1).
  - B returns to A: use a `bclr` instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

- **Indirect subroutine linkage:**
  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller; the Binder inserts "glue" code to mediate the branch.) The three branches should be as follows.
  - A calls Glue: use a `bl` or `bcl` instruction (LK=1).
  - Glue calls B: place the address of B into the Count Register, and use a `bcctr` instruction (LK=0).
  - B returns to A: use a `bclr` instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

- **Function call:**
  Here A calls a function, the identity of which may vary from one instance of the call to another, instead of calling a specific program B. This case should be handled using the conventions of the preceding two bullets, depending on whether the call is direct or indirect, with the following differences.
  - If the call is direct, place the address of the function into the Count Register, and use a `bcctr` instruction (LK=1) instead of a `bl` or `bcl` instruction.
  - For the `bcctr`[l] instruction that branches to the function, use BH=0b11 if appropriate.
The bits corresponding to the current “a” and “t” bits, and to the current “z” bits except in the “branch always” BO encoding, had different meanings in versions of the architecture that precede Version 2.00.

- The bit corresponding to the “t” bit was called the “y” bit. The “y” bit indicated whether to use the architected default prediction (y=0) or to use the complement of the default prediction (y=1). The default prediction was defined as follows.
  - If the instruction is $bc{l}[[a]$ with a negative value in the displacement field, the branch is taken. (This is the only case in which the prediction corresponding to the “y” bit differs from the prediction corresponding to the “t” bit.)
  - In all other cases ($bc{l}[[a]$ with a nonnegative value in the displacement field, $bclr{l}$, or $bcctr{l}$), the branch is not taken.

- The BO encodings that test both the Count Register and the Condition Register had a “y” bit in place of the current “z” bit. The meaning of the “y” bit was as described in the preceding item.

- The “a” bit was a “z” bit.

Because these bits have always been defined either to be ignored or to be treated as hints, a given program will produce the same result on any implementation regardless of the values of the bits. Also, because even the “y” bit is ignored, in practice, by most processors that comply with versions of the architecture that precede Version 2.00, the performance of a given program on those processors will not be affected by the values of the bits.
### Branch I-form

<table>
<thead>
<tr>
<th></th>
<th>target_addr</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>target_addr</td>
<td>(AA=0 LK=0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td>target_addr</td>
<td>(AA=1 LK=0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bl</td>
<td>target_addr</td>
<td>(AA=0 LK=1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bla</td>
<td>target_addr</td>
<td>(AA=1 LK=1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Branch Conditional

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>BO, BI, target_addr</td>
<td>(AA=0 LK=0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bca</td>
<td>BO, BI, target_addr</td>
<td>(AA=1 LK=0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcl</td>
<td>BO, BI, target_addr</td>
<td>(AA=0 LK=1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcla</td>
<td>BO, BI, target_addr</td>
<td>(AA=1 LK=1)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### B-form

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>LI</td>
<td>AA</td>
<td>LK</td>
<td>16</td>
<td>BO</td>
<td>BI</td>
<td>16</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- if AA then NIA $\leftarrow_{\text{lea}}$ EXTS(LI $||$ 0b00) $
- else NIA $\leftarrow_{\text{lea}}$ CIA + EXTS(LI $||$ 0b00) $
- if LK then LR $\leftarrow_{\text{lea}}$ CIA + 4 $

**target_addr** specifies the branch target address.

If AA=0 then the branch target address is the sum of LI $||$ 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI $||$ 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

- LR (if LK=1)

### Branch Conditional B-form

#### Extended Mnemonics:

- Examples of extended mnemonics for **Branch Conditional**:

  - **Extended:**
    - blt target
    - bne cr2,target
    - bdnz target

  - **Equivalent to:**
    - bc 12,0,target
    - bc 4,10,target
    - bc 16,0,target
Branch Conditional to Link Register

**XL-form**

<table>
<thead>
<tr>
<th>bclr</th>
<th>BO,BI,BH (LK=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bclrl</td>
<td>BO,BI,BH (LK=1)</td>
</tr>
</tbody>
</table>

if (64-bit mode)
then M ← 0
else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTR0:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRB1:32 = BO0)
if ctr_ok & cond_ok then NIA ← lea CTR0:61 || 0b00
if LK then LR ← lea CIA + 4

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 44. The BH field is used as described in Figure 46. The branch target address is LR0:61 || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

CTR (if BO₂=0)
LR (if LK=1)

**Extended Mnemonics:**

Examples of extended mnemonics for Branch Conditional to Link Register:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bclr</td>
<td>4,6</td>
</tr>
<tr>
<td>bclrl</td>
<td>4,6,0</td>
</tr>
<tr>
<td>btltlr</td>
<td>bclr 12,0,0</td>
</tr>
<tr>
<td>bnelr</td>
<td>bclr 4,10,0</td>
</tr>
<tr>
<td>bdnzlr</td>
<td>bclr 16,0,0</td>
</tr>
</tbody>
</table>

**Programming Note**

*bclr, bclrl, bcctr, and bcctrl* each serve as both a basic and an extended mnemonic. The Assembler will recognize a *bclr*, *bclrl*, *bcctr*, or *bcctrl* mnemonic with three operands as the basic form, and a *bclr*, *bclrl*, *bcctr*, or *bcctrl* mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00.

Branch Conditional to Count Register

**XL-form**

<table>
<thead>
<tr>
<th>bcctr</th>
<th>BO,BI,BH (LK=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcctrl</td>
<td>BO,BI,BH (LK=1)</td>
</tr>
</tbody>
</table>

cond_ok ← BO0 | (CRB1:32 = BO0)
if cond_ok then NIA ← lea CTR0:61 || 0b00
if LK then LR ← lea CIA + 4

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 44. The BH field is used as described in Figure 46. The branch target address is CTR0:61 || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

If the “decrement and test CTR” option is specified (BO₂=0), the instruction form is invalid.

**Special Registers Altered:**

LR (if LK=1)

**Extended Mnemonics:**

Examples of extended mnemonics for Branch Conditional to Count Register:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcctr</td>
<td>4,6</td>
</tr>
<tr>
<td>btctr</td>
<td>bcctr 12,0,0</td>
</tr>
<tr>
<td>bnectr</td>
<td>bcctr 4,10,0</td>
</tr>
<tr>
<td>bdnzlr</td>
<td>bcctr 16,0,0</td>
</tr>
</tbody>
</table>
**Branch Conditional to Branch Target Address Register**

**XL-form**

<table>
<thead>
<tr>
<th>bctar</th>
<th>BO, BI, BH (LK=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bctarl</td>
<td>BO, BI, BH (LK=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccccccc}
19 & 8 & 11 & /// & BH & 560 & LK \\
0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

if (64-bit mode)
then \( M \leftarrow 0 \)
else \( M \leftarrow 32 \)
if \( \neg B0_2 \) then \( CTR \leftarrow CTR - 1 \)
\( \text{ctr_ok} \leftarrow B0_2 \mid (\text{CTR}_{-63} \neq 0) \oplus B0_4 \)
\( \text{cond_ok} \leftarrow B0_0 \mid (\text{CTR}_{-32} = B0_2) \)
if \( \text{ctr_ok} \& \text{cond_ok} \) then \( \text{NIA} \leftarrow \text{iea TAR}_{0,61} \mid 0b00 \)
if \( LK \) then \( \text{LR} \leftarrow \text{iea CIA + 4} \)

\( \text{BI+32} \) specifies the Condition Register bit to be tested. The \( \text{BO} \) field is used to resolve the branch as described in Figure 44. The \( \text{BH} \) field is used as described in Figure 46. The branch target address is \( \text{TAR}_{0,61} \mid 0b00 \), with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If \( \text{LK=1} \) then the effective address of the instruction following the \text{Branch} instruction is placed into the \text{Link Register}.

**Special Registers Altered:**

- \( \text{CTR} \) (if \( B0_2=0 \))
- \( \text{LR} \) (if \( LK=1 \))

**Programming Note**

In some systems, the system program will restrict usage of the \text{bctar/1} instruction to only selected programs. If an attempt is made to execute the instruction when it is not available, the system error handler will be invoked. See Book III-S for additional information.
2.6 Condition Register Instructions

2.6.1 Condition Register Logical Instructions

The Condition Register Logical instructions have preferred forms; see Section 1.8.1. In the preferred forms, the BT and BB fields satisfy the following rule.

- The bit specified by BT is in the same Condition Register field as the bit specified by BB.

### Condition Register AND

**XL-form**

\[
\text{crand} \quad \text{BT,BA,BB}
\]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[\text{CR}_{BT+32} \leftarrow \text{CR}_{BA+32} \land \text{CR}_{BB+32}\]

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

### Condition Register NAND

**XL-form**

\[
\text{cmnand} \quad \text{BT,BA,BB}
\]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>225</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[\text{CR}_{BT+32} \leftarrow \neg(\text{CR}_{BA+32} \land \text{CR}_{BB+32})\]

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

### Condition Register OR

**XL-form**

\[
\text{cro} \quad \text{BT,BA,BB}
\]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>449</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[\text{CR}_{BT+32} \leftarrow \text{CR}_{BA+32} \lor \text{CR}_{BB+32}\]

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

### Condition Register XOR

**XL-form**

\[
\text{cxor} \quad \text{BT,BA,BB}
\]

<table>
<thead>
<tr>
<th>19</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>193</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[\text{CR}_{BT+32} \leftarrow \text{CR}_{BA+32} \oplus \text{CR}_{BB+32}\]

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

**Example of extended mnemonics for Condition Register OR:**

- **Extended:** crmove Bx,By
- **Equivalent to:** cro Bx,By,By

**Example of extended mnemonics for Condition Register XOR:**

- **Extended:** crcir Bx
- **Equivalent to:** cxor Bx,By,Bx

---

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics is provided that allow additional Condition Register logical operations, beyond those provided by the basic Condition Register Logical instructions, to be coded easily. Some of these are shown as examples with the Condition Register Logical instructions. See Appendix E for additional extended mnemonics.
### Condition Register NOR

**XL-form**

<table>
<thead>
<tr>
<th>crnor</th>
<th>BT, BA, BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 19 33 /</td>
<td>5 11 16 21</td>
</tr>
</tbody>
</table>

\[
\text{\text{CR}}_{BT+32} \leftarrow \neg (\text{CR}_{BA+32} \mid \text{CR}_{BB+32})
\]

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

**Extended Mnemonics:**

Example of extended mnemonics for Condition Register NOR:

- **Extended:** crnot Bx, By
- **Equivalent to:** crnor Bx, By, By

### Condition Register Equivalent

**XL-form**

<table>
<thead>
<tr>
<th>creqv</th>
<th>BT, BA, BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 19 289 /</td>
<td>5 11 16 21</td>
</tr>
</tbody>
</table>

\[
\text{\text{CR}}_{BT+32} \leftarrow \neg \text{CR}_{BA+32} = \text{CR}_{BB+32}
\]

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

**Extended Mnemonics:**

Example of extended mnemonics for Condition Register Equivalent:

- **Extended:** crset Bx
- **Equivalent to:** creqv Bx, Bx, Bx

### Condition Register AND with Complement

**XL-form**

<table>
<thead>
<tr>
<th>crandc</th>
<th>BT, BA, BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 19 129 /</td>
<td>5 11 16 21</td>
</tr>
</tbody>
</table>

\[
\text{\text{CR}}_{BT+32} \leftarrow \text{CR}_{BA+32} \& \neg \text{CR}_{BB+32}
\]

The bit in the Condition Register specified by BA+32 is ANDed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

### Condition Register OR with Complement

**XL-form**

<table>
<thead>
<tr>
<th>croc</th>
<th>BT, BA, BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 19 417 /</td>
<td>5 11 16 21</td>
</tr>
</tbody>
</table>

\[
\text{\text{CR}}_{BT+32} \leftarrow \text{CR}_{BA+32} \mid \neg \text{CR}_{BB+32}
\]

The bit in the Condition Register specified by BA+32 is ORed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[\text{CR}_{BT+32}\]

### 2.6.2 Condition Register Field Instruction

**Move Condition Register Field**

**XL-form**

<table>
<thead>
<tr>
<th>mcrf</th>
<th>BF, BFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 19 31 /</td>
<td>5 11 14 16 21</td>
</tr>
</tbody>
</table>

\[
\text{\text{CR}}_{4\times\text{BF+32}:4\times\text{BFA+35}} \leftarrow \text{\text{CR}}_{4\times\text{BF+32}:4\times\text{BFA+35}}
\]

The contents of Condition Register field BFA are copied to Condition Register field BF.

**Special Registers Altered:**

\[\text{CR field BF}\]
2.7 System Call Instruction

This instruction provides the means by which a program can call upon the system to perform a service.

<table>
<thead>
<tr>
<th><strong>System Call</strong></th>
<th><strong>SC-form</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>sc LEV</td>
<td>17 6 /// /// 11 16 // LEV // 1 // 27 30 31</td>
</tr>
</tbody>
</table>

This instruction calls the system to perform a service. A complete description of this instruction can be found in Book III.

The use of the LEV field is described in Book III. The LEV values greater than 1 are reserved, and bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

When control is returned to the program that executed the System Call instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

This instruction is context synchronizing (see Book III).

**Special Registers Altered:**
Dependent on the system service

**Programming Note**

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an sc mnemonic with one operand as the basic form, and an sc mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

In application programs the value of the LEV operand for sc should be 0.
2.8 Branch History Rolling Buffer Instructions

The Branch History Rolling Buffer instructions enable problem state programs to access and clear the BHRB. The availability of these instructions is controlled by the system program. (See Chapter 9 of Book III-S.) When an attempt is made to execute these instructions when they are unavailable, the Facility Unavailable error handler is invoked.

**Clear BHRB**

* X-form

```
clrhhrb
```

for \( n = 0:1023 \)

\( \text{BHRB}(n) \leftarrow 0 \)

All BHRB entries are set to 0s.

**Special Registers Altered:**

None.

---

**Move From Branch History Rolling Buffer**

* XFX-form

```
mbhhrbe RT,BHRBE
```

\( n \leftarrow \text{bhrbe}_0,9 \)

If \( n < \text{number of BHRBEs implemented} \), then

\( \text{RT} \leftarrow \text{BHRBE}(n) \)

else

\( \text{RT} \leftarrow 0 \)

The BHRBE field denotes an entry in the BHRB. If the designated entry is within the range of BHRB entries implemented, the contents of the designated BHRB entry are placed into register RT; otherwise, 0s are placed into RT.

In order to ensure that the current BHRB contents are read by this instruction, an event-based branch or rfebbb (see Chapter 7 of Book II) or a non-recoverable interrupt or Return from Interrupt instruction (see Book III-S Chapter 6) must have occurred prior to this instruction and after all previous Branch and clrbhrb instructions have completed.

**Special Registers Altered:**

None

---

**Programming Note**

In order to read all the BHRB entries containing information about taken branches, software reads the entries starting from entry number 0 and continuing until an entry containing 0s is read or until all implemented BHRB entries have been read.

---

**Programming Note**

Since the number of BHRB entries may decrease or the BHRB may be cleared at any time, reading an entry for a second time may result in a zero value being returned even though the program has not executed clrbhrb.
Chapter 3. Fixed-Point Facility

3.1 Fixed-Point Facility Overview

This chapter describes the registers and instructions that make up the Fixed-Point Facility.

3.2 Fixed-Point Facility Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Facility. The principal storage internal to the Fixed-Point Facility is a set of 32 General Purpose Registers (GPRs). See Figure 47.

![Figure 47. General Purpose Registers](image)

Each GPR is a 64-bit register.

3.2.2 Fixed-Point Exception Register

The Fixed-Point Exception Register (XER) is a 64-bit register.

![Figure 48. Fixed-Point Exception Register](image)

The bit definitions for the Fixed-Point Exception Register are shown below. Here \( M=0 \) in 64-bit mode and \( M=32 \) in 32-bit mode.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td><strong>Summary Overflow</strong> (SO)</td>
</tr>
<tr>
<td></td>
<td>The Summary Overflow bit is set to 1 whenever an instruction (except \textit{mtspr}) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an \textit{mtspr} instruction (specifying the XER) or an \textit{mcrxr} instruction. It is not altered by \textit{Compare} instructions, or by other instructions (except \textit{mtspr} to the XER, and \textit{mcrxr}) that cannot overflow. Executing an \textit{mtspr} instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.</td>
</tr>
<tr>
<td>33</td>
<td><strong>Overflow</strong> (OV)</td>
</tr>
</tbody>
</table>
|        | The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. \( \text{XO-form Add, Subtract From, and Negate} \) instructions having OE=1 set it to 1 if the carry out of bit \( M \) is not equal to the carry out of bit \( M+1 \), and set it to 0 otherwise. \( \text{XO-form Multiply Low and Divide} \) instructions having OE=1 set it to 1 if the result cannot be represented in 64 bits (\textit{mulld, divd, divid, divdu, divdeu}) or in 32 bits (\textit{mulw, divw, divwe, divwu, divweu}), and set it to 0 otherwise. The OV bit is not altered by \textit{Compare} instructions.
instructions, or by other instructions (except \texttt{mtspr} to the XER, and \texttt{mcrxr}) that cannot overflow.

[Category: Legacy Integer Multiply-Accumulate]  
XO-form \textit{Legacy Integer Multiply-Accumulate} instructions set OV when OE=1 to reflect overflow of the 32-bit result. For signed-integer accumulation, overflow occurs when the add produces a carry out of bit 32 that is not equal to the carry out of bit 33. For unsigned-integer accumulation, overflow occurs when the add produces a carry out of bit 32.

\textbf{Carry (CA)}

The Carry bit is set as follows, during execution of certain instructions. \textit{Add Carrying}, \textit{Subtract From Carrying}, \textit{Add Extended}, and \textit{Subtract From Extended} types of instructions set it to 1 if there is a carry out of bit M, and set it to 0 otherwise. \textit{Shift Right Algebraic} instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by \textit{Compare} instructions, or by other instructions (except \textit{Shift Right Algebraic}, \texttt{mtspr} to the XER, and \texttt{mcrxr}) that cannot carry.

34:56 Reserved

57:63 This field specifies the number of bytes to be transferred by a \textit{Load String Indexed} or \textit{Store String Indexed} instruction.

[Category: Legacy Move Assist]

This field is used as a target by \texttt{dmlzb} to indicate the byte location of the leftmost zero byte found.

### 3.2.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register that can be used as a software use SPR; see Sections 3.2.4 and 5.3.3.

#### Architecture Note

In versions of the Architecture that precede Version 2.05, the VRSAVE Register was in the Vector category (see Section 5.3.3). It is now included in the Base category to simplify operating system support of and migration between heterogeneous systems containing processors with and without support for the Vector category.

### 3.2.4 Software Use SPRs [Category: Embedded]

Software Use SPRs are 64-bit registers that have no defined functionality. SPRG4-7 can be read by application programs. Additional Software Use SPRs are defined in Book III.

<table>
<thead>
<tr>
<th>SPRG4</th>
<th>SPRG5</th>
<th>SPRG6</th>
<th>SPRG7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

\textbf{Figure 49. Software-use SPRs}

\textbf{Programming Note}

USPRG0 was made a 32-bit register and renamed to VRSAVE; see Sections 3.2.3 and 5.3.3.

### 3.2.5 Device Control Registers [Category: Embedded.Device Control]

Device Control Registers (DCRs) are on-chip registers that exist architecturally outside the processor and thus are not actually part of the processor architecture. This specification simply defines the existence of a Device Control Register ‘address space’ and the instructions to access them and does not define the Device Control Registers themselves.

Device Control Registers may control the use of on-chip peripherals, such as memory controllers (the definition of specific Device Control Registers is implementation-dependent).

The contents of user-mode-accessible Device Control Registers can be read using \texttt{mfdcrux} and written using \texttt{mtdcrux}.
3.3 Fixed-Point Facility Instructions

3.3.1 Fixed-Point Storage Access Instructions

The Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3 on page 26.

Programming Note

The ia extended mnemonic permits computing an effective address as a Load or Store instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address.

3.3.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.

3.3.2 Fixed-Point Load Instructions

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into register RT.

Many of the Load instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into register RA and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT.

Programming Note

In some implementations, the Load Algebraic and Load with Update instructions may have greater latency than other types of Load instructions. Moreover, Load with Update instructions may take longer to execute in some implementations than the corresponding pair of a non-update Load instruction and an Add instruction.
### Load Byte and Zero

**D-form**

\[
\text{lbz} \quad RT, D(RA)
\]

<table>
<thead>
<tr>
<th>0</th>
<th>34</th>
<th>RT</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + \text{EXTS}(D) \)
\( RT \leftarrow 560 \| \text{MEM}(EA, 1) \)

Let the effective address (EA) be the sum \((RA|0)+ D\). The byte in storage addressed by EA is loaded into \( RT_{56:63} \). \( RT_{0:55} \) are set to 0.

**Special Registers Altered:**

None

---

### Load Byte and Zero Indexed

**X-form**

\[
\text{lbzx} \quad RT, RA, RB
\]

| 0 | 31 | RT | 11 | RA | 16 | RB | 87 | /
|---|----|----|----|----|----|----|----|-----|

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 560 \| \text{MEM}(EA, 1) \)

Let the effective address (EA) be the sum \((RA|0)+ (RB)\). The byte in storage addressed by EA is loaded into \( RT_{56:63} \). \( RT_{0:55} \) are set to 0.

**Special Registers Altered:**

None

---

### Load Byte and Zero with Update

**D-form**

\[
\text{lbzu} \quad RT, D(RA)
\]

<table>
<thead>
<tr>
<th>0</th>
<th>35</th>
<th>RT</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

\( EA \leftarrow (RA) + \text{EXTS}(D) \)
\( RT \leftarrow 560 \| \text{MEM}(EA, 1) \)
\( RA \leftarrow EA \)

Let the effective address (EA) be the sum \((RA)+ D\). The byte in storage addressed by EA is loaded into \( RT_{56:63} \). \( RT_{0:55} \) are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**

None

---

### Load Byte and Zero with Update Indexed

**X-form**

\[
\text{lbzux} \quad RT, RA, RB
\]

| 0 | 31 | RT | 11 | RA | 16 | RB | 119 | /
|---|----|----|----|----|----|----|-----|-----|

\( EA \leftarrow (RA) + (RB) \)
\( RT \leftarrow 560 \| \text{MEM}(EA, 1) \)
\( RA \leftarrow EA \)

Let the effective address (EA) be the sum \((RA)+(RB)\). The byte in storage addressed by EA is loaded into \( RT_{56:63} \). \( RT_{0:55} \) are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**

None
### Load Halfword and Zero

**D-form**

\[
\text{lhz} \quad \text{RT}, \text{D(RA)}
\]

<table>
<thead>
<tr>
<th>D-form</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lhz} \quad \text{RT}, \text{D(RA)} )</td>
<td>( \text{lhz} \quad \text{RT}, \text{RA}, \text{RB} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>40</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

- if RA = 0 then \( b \leftarrow 0 \)
- else \( b \leftarrow \text{RA} \)
- \( \text{EA} \leftarrow b + \text{EXTS(D)} \)
- \( \text{RT} \leftarrow \text{RT}0:47 \parallel \text{MEM(EA, 2)} \)

Let the effective address (EA) be the sum \((\text{RA}) + \text{D}\). The halfword in storage addressed by EA is loaded into RT\(_{48:63}\), RT\(_{0:47}\) are set to 0.

**Special Registers Altered:**
- None

### Load Halfword and Zero Indexed

**X-form**

\[
\text{lhzx} \quad \text{RT}, \text{RA}, \text{RB}
\]

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lhzx} \quad \text{RT}, \text{RA}, \text{RB} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

- if RA = 0 then \( b \leftarrow 0 \)
- else \( b \leftarrow \text{RA} \)
- \( \text{EA} \leftarrow b + \text{(RB)} \)
- \( \text{RT} \leftarrow \text{RT}0:47 \parallel \text{MEM(EA, 2)} \)

Let the effective address (EA) be the sum \((\text{RA0}) + \text{(RB)}\). The halfword in storage addressed by EA is loaded into RT\(_{48:63}\), RT\(_{0:47}\) are set to 0.

**Special Registers Altered:**
- None

### Load Halfword and Zero with Update

**D-form**

\[
\text{lhzu} \quad \text{RT}, \text{D(RA)}
\]

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lhzu} \quad \text{RT}, \text{D(RA)} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>41</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

- \( \text{EA} \leftarrow (\text{RA}) + \text{EXTS(D)} \)
- \( \text{RT} \leftarrow \text{RT}0:47 \parallel \text{MEM(EA, 2)} \)
- \( \text{RA} \leftarrow \text{EA} \)

Let the effective address (EA) be the sum \((\text{RA}) + \text{D}\). The halfword in storage addressed by EA is loaded into RT\(_{48:63}\), RT\(_{0:47}\) are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**
- None

### Load Halfword and Zero with Update Indexed

**X-form**

\[
\text{lhuzx} \quad \text{RT}, \text{RA}, \text{RB}
\]

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lhuzx} \quad \text{RT}, \text{RA}, \text{RB} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

- \( \text{EA} \leftarrow (\text{RA}) + \text{(RB)} \)
- \( \text{RT} \leftarrow \text{RT}0:47 \parallel \text{MEM(EA, 2)} \)
- \( \text{RA} \leftarrow \text{EA} \)

Let the effective address (EA) be the sum \((\text{RA}) + \text{(RB)}\). The halfword in storage addressed by EA is loaded into RT\(_{48:63}\), RT\(_{0:47}\) are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**
- None
Load Halfword Algebraic

\[ \text{lhax} \quad RT, D(RA) \]

<table>
<thead>
<tr>
<th>42</th>
<th>RT</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{if } RA = 0 & \text{ then } b \leftarrow 0 \\
\text{else} & \quad b \leftarrow (RA) \\
EA & \leftarrow b + \text{EXTS(D)} \\
RT & \leftarrow \text{EXTS(MEM(EA, 2))}
\end{align*}
\]

Let the effective address (EA) be the sum (RA|0)+ D. The halfword in storage addressed by EA is loaded into \(RT_{48:63}\). \(RT_{0:47}\) are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:
None

Load Halfword Algebraic Indexed \(X\)-form

\[ \text{lhax} \quad RT, RA, RB \]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>343</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{if } RA = 0 & \text{ then } b \leftarrow 0 \\
\text{else} & \quad b \leftarrow (RA) \\
EA & \leftarrow b + (RB) \\
RT & \leftarrow \text{EXTS(MEM(EA, 2))}
\end{align*}
\]

Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into \(RT_{48:63}\). \(RT_{0:47}\) are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:
None

Load Halfword Algebraic with Update \(D\)-form

\[ \text{lhau} \quad RT, D(RA) \]

<table>
<thead>
<tr>
<th>43</th>
<th>RT</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
EA & \leftarrow (RA) + \text{EXTS(D)} \\
RT & \leftarrow \text{EXTS(MEM(EA, 2))} \\
RA & \leftarrow EA
\end{align*}
\]

Let the effective address (EA) be the sum (RA)+ D. The halfword in storage addressed by EA is loaded into \(RT_{48:63}\). \(RT_{0:47}\) are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword Algebraic with Update Indexed \(X\)-form

\[ \text{lhau} \quad RT, RA, RB \]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>375</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
EA & \leftarrow (RA) + (RB) \\
RT & \leftarrow \text{EXTS(MEM(EA, 2))} \\
RA & \leftarrow EA
\end{align*}
\]

Let the effective address (EA) be the sum (RA)+(RB). The halfword in storage addressed by EA is loaded into \(RT_{48:63}\). \(RT_{0:47}\) are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None
### Load Word and Zero 

**D-form**

\[
lwz \quad RT, D(RA)
\]

<table>
<thead>
<tr>
<th>32</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

If RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
EA \( \leftarrow b + \text{EXTS}(D) \)
RT \( \leftarrow 32_0 || \text{MEM}(EA, 4) \)

Let the effective address (EA) be the sum (RA|0)+ D. The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are set to 0.

**Special Registers Altered:** None

### Load Word and Zero Indexed 

**X-form**

\[
lwzx \quad RT, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>23</th>
<th>31</th>
</tr>
</thead>
</table>

If RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
EA \( \leftarrow b + (RB) \)
RT \( \leftarrow 32_0 || \text{MEM}(EA, 4) \)

Let the effective address (EA) be the sum (RA0)+ (RB). The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are set to 0.

**Special Registers Altered:** None

### Load Word and Zero with Update 

**D-form**

\[
lwzu \quad RT, D(RA)
\]

<table>
<thead>
<tr>
<th>33</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

EA \( \leftarrow (RA) + \text{EXTS}(D) \)
RT \( \leftarrow 32_0 || \text{MEM}(EA, 4) \)
RA \( \leftarrow EA \)

Let the effective address (EA) be the sum (RA)+ D. The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:** None

### Load Word and Zero with Update Indexed 

**X-form**

\[
lwzux \quad RT, RA, RB
\]

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>55</th>
<th>31</th>
</tr>
</thead>
</table>

EA \( \leftarrow (RA) + (RB) \)
RT \( \leftarrow 32_0 || \text{MEM}(EA, 4) \)
RA \( \leftarrow EA \)

Let the effective address (EA) be the sum (RA)+ (RB). The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:** None
3.3.2.1 64-bit Fixed-Point Load Instructions [Category: 64-Bit]

### Load Word Algebraic

**DS-form**

\[
lwa_{DS} \quad RT, DS(RA)
\]

<table>
<thead>
<tr>
<th></th>
<th>58</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RT</td>
<td>RA</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\[
EA \leftarrow b + \text{EXTS}(DS \| 0b00)
\]

\[
RT \leftarrow \text{EXTS}(\text{MEM}(EA, 4))
\]

Let the effective address (EA) be the sum \((RA|0)+(DS||0b00)\). The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are filled with a copy of bit 0 of the loaded word.

**Special Registers Altered:**

None

### Load Word Algebraic Indexed

**X-form**

\[
lwa_{X} \quad RT, RA, RB
\]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\[
EA \leftarrow b + (RB)
\]

\[
RT \leftarrow \text{EXTS}(\text{MEM}(EA, 4))
\]

Let the effective address (EA) be the sum \((RA|0)+(RB)\). The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are filled with a copy of bit 0 of the loaded word.

**Special Registers Altered:**

None

### Load Word Algebraic with Update Indexed

**X-form**

\[
lwa_{U} \quad RT, RA, RB
\]

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>373</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
EA \leftarrow (RA) + (RB)
\]

\[
RT \leftarrow \text{EXTS}(\text{MEM}(EA, 4))
\]

\[
RA \leftarrow EA
\]

Let the effective address (EA) be the sum \((RA)+(RB)\). The word in storage addressed by EA is loaded into RT\(_{32:63}\). RT\(_{0:31}\) are filled with a copy of bit 0 of the loaded word.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

**Special Registers Altered:**

None
### Load Doubleword  DS-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld RT,DS(RA)</code></td>
<td>Load Doubleword from DS to RT.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DS</code></td>
<td>Data Segment</td>
</tr>
<tr>
<td><code>RT</code></td>
<td>Register Target</td>
</tr>
<tr>
<td><code>RA</code></td>
<td>Register Address</td>
</tr>
</tbody>
</table>

#### Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld RT,DS(0)</code></td>
<td>Load Doubleword from DS(0) to RT.</td>
</tr>
</tbody>
</table>

#### Implementation Details

If RA = 0 then b ← 0
else b ← (RA)

EA ← b + EXTS(DS || 0b00)
RT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0) + (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

**Special Registers Altered:** None

### Load Doubleword Indexed  X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldx RT,RA,RB</code></td>
<td>Load Doubleword Indexed to RT.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X</code></td>
<td>Indexed Access</td>
</tr>
<tr>
<td><code>RT</code></td>
<td>Register Target</td>
</tr>
<tr>
<td><code>RA</code></td>
<td>Register Address</td>
</tr>
<tr>
<td><code>RB</code></td>
<td>Register Base</td>
</tr>
</tbody>
</table>

#### Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldx RT,0,0</code></td>
<td>Load Doubleword Indexed from 0,0 to RT.</td>
</tr>
</tbody>
</table>

#### Implementation Details

If RA = 0 then b ← 0
else b ← (RA)

EA ← b + (RB)
RT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0) + (RB). The doubleword in storage addressed by EA is loaded into RT.

**Special Registers Altered:** None

### Load Doubleword with Update  DS-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldu RT,DS(RA)</code></td>
<td>Load Doubleword with Update from DS to RT.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DS</code></td>
<td>Data Segment</td>
</tr>
<tr>
<td><code>RT</code></td>
<td>Register Target</td>
</tr>
<tr>
<td><code>RA</code></td>
<td>Register Address</td>
</tr>
</tbody>
</table>

#### Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldu RT,DS(0)</code></td>
<td>Load Doubleword with Update from DS(0) to RT.</td>
</tr>
</tbody>
</table>

#### Implementation Details

EA ← (RA) + EXTS(DS || 0b00)
RT ← MEM(EA, 8)
RA ← EA

Let the effective address (EA) be the sum (RA|0) + (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

**Special Registers Altered:** None

### Load Doubleword with Update Indexed  X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldux RT,RA,RB</code></td>
<td>Load Doubleword with Update Indexed to RT.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X</code></td>
<td>Indexed Access</td>
</tr>
<tr>
<td><code>RT</code></td>
<td>Register Target</td>
</tr>
<tr>
<td><code>RA</code></td>
<td>Register Address</td>
</tr>
<tr>
<td><code>RB</code></td>
<td>Register Base</td>
</tr>
</tbody>
</table>

#### Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldux RT,0,0</code></td>
<td>Load Doubleword with Update Indexed from 0,0 to RT.</td>
</tr>
</tbody>
</table>

#### Implementation Details

EA ← (RA) + (RB)
RT ← MEM(EA, 8)
RA ← EA

Let the effective address (EA) be the sum (RA|0) + (RB). The doubleword in storage addressed by EA is loaded into RT.

**Special Registers Altered:** None

**Notes:**
If RA=0 or RA=RT, the instruction form is invalid.
3.3.3 Fixed-Point Store Instructions

The contents of register RS are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Many of the Store instructions have an “update” form, in which register RA is updated with the effective address. For these forms, the following rules apply.

- If RA ≠ 0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target storage element and then EA is placed into RA (RS).

### Store Byte

<table>
<thead>
<tr>
<th>D-form</th>
<th>Store Byte Indexed</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>stb</td>
<td>RS,D(RA)</td>
<td>stbx</td>
</tr>
<tr>
<td>38</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + EXTS(D)  
MEM(EA, 1) ← (RS)56:63  

Let the effective address (EA) be the sum (RA|0)+ D. (RS)56:63 are stored into the byte in storage addressed by EA.

Special Registers Altered:
None

### Store Byte with Update

<table>
<thead>
<tr>
<th>D-form</th>
<th>Store Byte with Update Indexed</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>stbu</td>
<td>RS,D(RA)</td>
<td>stbux</td>
</tr>
<tr>
<td>39</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(D)  
MEM(EA, 1) ← (RS)56:63  
RA ← EA  

Let the effective address (EA) be the sum (RA)+ D. (RS)56:63 are stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None
### Store Halfword

**D-form**

\[ \text{sth} \quad RS, D(RA) \]

<table>
<thead>
<tr>
<th>D</th>
<th>44</th>
<th>RS</th>
<th>RA</th>
<th>16</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + EXTS(D)  
MEM(EA, 2) ← (RS)_{48:63}

Let the effective address (EA) be the sum (RA|0)+ D. (RS)_{48:63} are stored into the halfword in storage addressed by EA.

**Special Registers Altered:**
None

### Store Halfword Indexed

**X-form**

\[ \text{sthx} \quad RS, RA, RB \]

<table>
<thead>
<tr>
<th>D</th>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>16</th>
<th>21</th>
<th>407</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
MEM(EA, 2) ← (RS)_{48:63}

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS)_{48:63} are stored into the halfword in storage addressed by EA.

**Special Registers Altered:**
None

### Store Halfword with Update

**D-form**

\[ \text{sthu} \quad RS, D(RA) \]

<table>
<thead>
<tr>
<th>D</th>
<th>45</th>
<th>RS</th>
<th>RA</th>
<th>16</th>
</tr>
</thead>
</table>

EA ← (RA) + EXTS(D)  
MEM(EA, 2) ← (RS)_{48:63}  
RA ← EA

Let the effective address (EA) be the sum (RA)+ D. (RS)_{48:63} are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None

### Store Halfword with Update Indexed

**X-form**

\[ \text{sthux} \quad RS, RA, RB \]

<table>
<thead>
<tr>
<th>D</th>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>16</th>
<th>21</th>
<th>439</th>
<th>31</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)  
MEM(EA, 2) ← (RS)_{48:63}  
RA ← EA

Let the effective address (EA) be the sum (RA)+ (RB). (RS)_{48:63} are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None
Store Word

\texttt{stw RS,D(RA)}

\begin{verbatim}
0 36 11 16 D
\end{verbatim}

\begin{itemize}
\item if RA = 0 then b \leftarrow 0
\item else \quad b \leftarrow (RA)
\item EA \leftarrow b + \text{EXTS(D)}
\item \text{MEM}(EA, 4) \leftarrow (RS)_{32:63}
\end{itemize}

Let the effective address (EA) be the sum (RA|0)+ D. (RS)_{32:63} are stored into the word in storage addressed by EA.

Special Registers Altered:
None

<table>
<thead>
<tr>
<th>Store Word Indexed</th>
</tr>
</thead>
</table>

\texttt{stwx RS,RA,RB}

\begin{verbatim}
0 31 11 16 21 151 / 31
\end{verbatim}

\begin{itemize}
\item if RA = 0 then b \leftarrow 0
\item else \quad b \leftarrow (RA)
\item EA \leftarrow b + (RB)
\item \text{MEM}(EA, 4) \leftarrow (RS)_{32:63}
\end{itemize}

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS)_{32:63} are stored into the word in storage addressed by EA.

Special Registers Altered:
None

<table>
<thead>
<tr>
<th>Store Word with Update</th>
</tr>
</thead>
</table>

\texttt{stwu RS,D(RA)}

\begin{verbatim}
0 37 11 16 D
\end{verbatim}

\begin{itemize}
\item EA \leftarrow (RA) + \text{EXTS(D)}
\item \text{MEM}(EA, 4) \leftarrow (RS)_{32:63}
\item RA \leftarrow EA
\end{itemize}

Let the effective address (EA) be the sum (RA)+ D. (RS)_{32:63} are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

<table>
<thead>
<tr>
<th>Store Word with Update Indexed</th>
</tr>
</thead>
</table>

\texttt{stwux RS,RA,RB}

\begin{verbatim}
0 31 11 16 21 183 / 31
\end{verbatim}

\begin{itemize}
\item EA \leftarrow (RA) + (RB)
\item \text{MEM}(EA, 4) \leftarrow (RS)_{32:63}
\item RA \leftarrow EA
\end{itemize}

Let the effective address (EA) be the sum (RA)+ (RB). (RS)_{32:63} are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None
3.3.3.1 64-bit Fixed-Point Store Instructions [Category: 64-Bit]

**Store Doubleword**

<table>
<thead>
<tr>
<th>DS-form</th>
<th>std RS,DS(RA)</th>
</tr>
</thead>
</table>

| 62 RS RA DS 0 |
| 0 6 11 16 30 31 |

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

**Special Registers Altered:**
None

**Store Doubleword Indexed**

<table>
<thead>
<tr>
<th>X-form</th>
<th>stdx RS,RA,RB</th>
</tr>
</thead>
</table>

| 31 RS RA RB 149 7 |
| 0 6 11 16 21 31 |

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)

Let the effective address (EA) be the sum (RA0)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

**Special Registers Altered:**
None

**Store Doubleword with Update**

<table>
<thead>
<tr>
<th>DS-form</th>
<th>stdu RS,DS(RA)</th>
</tr>
</thead>
</table>

| 62 RS RA DS 1 |
| 0 6 11 16 30 31 |

EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA

Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

**Special Registers Altered:**
None

**Store Doubleword with Update Indexed**

<table>
<thead>
<tr>
<th>X-form</th>
<th>stdux RS,RA,RB</th>
</tr>
</thead>
</table>

| 31 RS RA RB 181 7 |
| 0 6 11 16 21 31 |

EA ← (RA) + (RB)
MEM(EA, 8) ← (RS)
RA ← EA

Let the effective address (EA) be the sum (RA)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

**Special Registers Altered:**
None
3.3.4 Fixed-Point Load and Store Quadword Instructions [Category: Load/Store Quadword]

**Load Quadword**

<table>
<thead>
<tr>
<th>lq</th>
<th>RTp,DQ(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>RTp</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>DQ</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(DQ || 0b0000)
RTp ← MEM(EA, 16)

Let the effective address (EA) be the sum (RA|0)+(DQ||0b0000). The quadword in storage addressed by EA is loaded into register pair RTp.

If RTp is odd or RTp=RA, the instruction form is invalid. If RTp=RA, an attempt to execute this instruction causes an Illegal Instruction type Program interrupt. (The RTp=RA case includes the case of RTp=RA=0.)

The quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In big-endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

**Store Quadword**

<table>
<thead>
<tr>
<th>stq</th>
<th>RSp,DS(RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>RSp</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>DS</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 16) ← RSp

Let the effective address (EA) be the sum (RA|0)+(DS||0b00). The contents of register pair RSp are stored into the quadword in storage addressed by EA.

If RSp is odd, the instruction form is invalid.

The contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In big-endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

**Special Registers Altered:**
None
### 3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions

**Programming Note**

These instructions have the effect of loading and storing data in the opposite byte ordering from that which would be used by other Load and Store instructions.

---

#### Load Halfword Byte-Reverse Indexed

**X-form**

lhbrx RT,RA,RB

| 31 | 6 | 11 | 16 | 21 | 790 | /
|----|---|----|----|----|-----|---
| 0  |   |    |    |    |     | 31

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
load_data ← MEM(EA, 2)  
RT ← \[ \text{load\_data}_{2:0} || \text{load\_data}_{10:8} \]

Let the effective address (EA) be the sum (RA|0)+ (RB). Bits 0:7 of the halfword in storage addressed by EA are loaded into RT\[56:63\]. Bits 8:15 of the halfword in storage addressed by EA are loaded into RT\[48:55\]. RT\[0:47\] are set to 0.

**Special Registers Altered:**

None

---

#### Store Halfword Byte-Reverse Indexed

**X-form**

sthbrx RS,RA,RB

| 31 | 6 | 11 | 16 | 21 | 918 | /
|----|---|----|----|----|-----|---
| 0  |   |    |    |    |     | 31

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
MEM(EA, 2) ← (RS)56:63 || (RS)48:55

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS)56:63 are stored into bits 0:7 of the halfword in storage addressed by EA. (RS)48:55 are stored into bits 8:15 of the halfword in storage addressed by EA.

**Special Registers Altered:**

None

---

#### Load Word Byte-Reverse Indexed

**X-form**

lwbrx RT,RA,RB

| 31 | 6 | 11 | 16 | 21 | 534 | /
|----|---|----|----|----|-----|---
| 0  |   |    |    |    |     | 31

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
load_data ← MEM(EA, 4)  
RT ← \[ \text{load\_data}_{24:31} || \text{load\_data}_{16:23} \]

Let the effective address (EA) be the sum (RA|0)+ (RB). Bits 0:7 of the word in storage addressed by EA are loaded into RT\[56:63\]. Bits 8:15 of the word in storage addressed by EA are loaded into RT\[48:55\]. Bits 16:23 of the word in storage addressed by EA are loaded into RT\[40:47\]. Bits 24:31 of the word in storage addressed by EA are loaded into RT\[32:39\]. RT\[0:31\] are set to 0.

**Special Registers Altered:**

None

---

#### Store Word Byte-Reverse Indexed

**X-form**

stwcharx RS,RA,RB

| 31 | 6 | 11 | 16 | 21 | 662 | /
|----|---|----|----|----|-----|---
| 0  |   |    |    |    |     | 31

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
MEM(EA, 4) ← (RS)56:63 || (RS)48:55 || (RS)40:47

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS)56:63 are stored into bits 0:7 of the word in storage addressed by EA. (RS)48:55 are stored into bits 8:15 of the word in storage addressed by EA. (RS)40:47 are stored into bits 16:23 of the word in storage addressed by EA. (RS)32:39 are stored into bits 24:31 of the word in storage addressed by EA.

**Special Registers Altered:**

None

---

**Programming Note**

In some implementations, the Load Byte-Reverse instructions may have greater latency than other Load instructions.
3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions [Category: 64-bit]

**Load Doubleword Byte-Reverse Indexed X-form**

```plaintext
ldbrx RT,RA,RB

if RA = 0 then b ← 0
else b ← {RA}
EA ← b + {RB}
load_data ← MEM(EA, 8)
RT ← load_data56:63 || load_data48:55
  | | load_data40:47 || load_data32:39
  | | load_data24:31 || load_data16:23
  | | load_data16:23 || load_data8:15

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the doubleword in storage addressed by EA are loaded into RT56:63. Bits 8:15 of the doubleword in storage addressed by EA are loaded into RT48:55. Bits 16:23 of the doubleword in storage addressed by EA are loaded into RT40:47. Bits 24:31 of the doubleword in storage addressed by EA are loaded into RT32:39. Bits 32:39 of the doubleword in storage addressed by EA are loaded into RT24:31. Bits 40:47 of the doubleword in storage addressed by EA are loaded into RT16:23. Bits 48:55 of the doubleword in storage addressed by EA are loaded into RT8:15. Bits 56:63 of the doubleword in storage addressed by EA are loaded into RT0:7.

**Special Registers Altered:**

None
```

**Store Doubleword Byte-Reverse Indexed X-form**

```plaintext
stdbrx RS,RA,RB

if RA = 0 then b ← 0
else b ← {RA}
EA ← b + {RB}
MEM(EA, 8) ← (RS)56:63 || (RS)48:55
  | | (RS)40:47 || (RS)32:39
  | | (RS)24:31 || (RS)16:23
  | | (RS)16:23 || (RS)8:15
  | | (RS)8:15 || (RS)0:7

Let the effective address (EA) be the sum (RA|0)+(RB). (RS)56:63 are stored into bits 0:7 of the doubleword in storage addressed by EA. (RS)48:55 are stored into bits 8:15 of the doubleword in storage addressed by EA. (RS)40:47 are stored into bits 16:23 of the doubleword in storage addressed by EA. (RS)32:39 are stored into bits 23:31 of the doubleword in storage addressed by EA. (RS)24:31 are stored into bits 32:39 of the doubleword in storage addressed by EA. (RS)16:23 are stored into bits 40:47 of the doubleword in storage addressed by EA. (RS)8:15 are stored into bits 48:55 of the doubleword in storage addressed by EA. (RS)0:7 are stored into bits 56:63 of the doubleword in storage addressed by EA.

**Special Registers Altered:**

None
3.3.6 Fixed-Point Load and Store Multiple Instructions

The *Load/Store Multiple* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 22. In the preferred forms, storage alignment satisfies the following rule.

The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

For the Server environment, the *Load/Store Multiple* instructions are not supported in Little-Endian mode. If they are executed in Little-Endian mode, the system alignment error handler is invoked.

---

**Load Multiple Word**

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="D-form" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load Multiple Word</th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lmw RT,D(RA)</code></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
r ← RT
do while r ≤ 31
    GPR[r] ← 32:0 || MEM(EA, 4)
r ← r + 1
    EA ← EA + 4

Let n = (32–RT). Let the effective address (EA) be the sum (RA|0)+ D.

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

**Special Registers Altered:**

None

---

**Store Multiple Word**

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="D-form" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store Multiple Word</th>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stmw RS,D(RA)</code></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
r ← RS
do while r ≤ 31
    MEM(EA, 4) ← GPR[r]32:63
    r ← r + 1
    EA ← EA + 4

Let n = (32–RS). Let the effective address (EA) be the sum (RA|0)+ D.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

**Special Registers Altered:**

None
3.3.7 Fixed-Point Move Assist Instructions [Category: Move Assist]

The *Move Assist* instructions allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

The *Load/Store String* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 22. In the preferred forms, register usage satisfies the following rules.

- RS = 4 or 5
- RT = 4 or 5
- Last register loaded/stored ≤ 12

For some implementations, using GPR 4 for RS and RT may result in slightly faster execution than using GPR 5.

For the Server environment, the *Move Assist* instructions are not supported in Little-Endian mode. If they are executed in Little-Endian mode, the system alignment error handler may be invoked or the instructions may be treated as no-ops if the number of bytes specified by the instruction is 0.
Load String Word Immediate  
\[ \text{lswi } \text{RT,RA,NB} \]

\begin{array}{cccccccc}
\hline
31 & 8 & 11 & 16 & 21 & 597 & \backslash \\
\hline
0 & \text{RT} & \text{RA} & \text{NB} & \text{n} & \text{EA} & \text{n} & \text{r} & \text{i}\backslash \\
\hline
\end{array}

\begin{itemize}
  \item if RA = 0 then EA \leftarrow 0
  \item else EA \leftarrow (RA)
  \item if NB = 0 then n \leftarrow 32
  \item else n \leftarrow NB
  \item r \leftarrow RT - 1
  \item i \leftarrow 32
  \item do while n > 0
    \begin{itemize}
      \item if i = 32 then
        \begin{itemize}
          \item r \leftarrow r + 1 (mod 32)
          \item GPR(r) \leftarrow 0
          \item GPR(r)_{i:i+7} \leftarrow MEM(EA, 1)
          \item i \leftarrow i + 8
        \end{itemize}
      \item if i = 64 then i \leftarrow 32
      \item EA \leftarrow EA + 1
      \item n \leftarrow n - 1
    \end{itemize}
  \end{itemize}
\end{itemize}

Let the effective address (EA) be (RA0). Let n = NB if NB\neq 0, n = 32 if NB=0; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load String Word Indexed  
\[ \text{lswx } \text{RT,RA,RB} \]

\begin{array}{cccccccc}
\hline
31 & 8 & 11 & 16 & 21 & 533 & \backslash \\
\hline
0 & \text{RT} & \text{RA} & \text{RB} & \text{n} & \text{EA} & \text{n} & \text{r} & \text{i}\backslash \\
\hline
\end{array}

\begin{itemize}
  \item if RA = 0 then b \leftarrow 0
  \item else b \leftarrow (RA)
  \item EA \leftarrow b + (RB)
  \item n \leftarrow XER_{57:63}
  \item r \leftarrow RT - 1
  \item i \leftarrow 32
  \item RT \leftarrow undefined
  \item do while n > 0
    \begin{itemize}
      \item if i = 32 then
        \begin{itemize}
          \item r \leftarrow r + 1 (mod 32)
          \item GPR(r) \leftarrow 0
          \item GPR(r)_{i:i+7} \leftarrow MEM(EA, 1)
          \item i \leftarrow i + 8
        \end{itemize}
      \item if i = 64 then i \leftarrow 32
      \item EA \leftarrow EA + 1
      \item n \leftarrow n - 1
    \end{itemize}
  \end{itemize}
\end{itemize}

Let the effective address (EA) be the sum (RA0)+(RB). Let n=XER_{57:63}; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, the instruction is treated as if the instruction form were invalid. If RT=RA or RT=RB, the instruction form is invalid.

Special Registers Altered:
None
Store String Word Immediate  X-form

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>NB</th>
<th>725</th>
<th>/</th>
</tr>
</thead>
</table>

stswi RS,RA,NB

if RA = 0 then EA ← 0
else           EA ← (RA)

if NB = 0 then n ← 32
else           n ← NB

r ← RS - 1
i ← 32

do while n > 0
    if i = 32 then r ← r + 1 (mod 32)
    MEM(EA, 1) ← GPR(r)i:i+7
    i ← i + 8
    if i = 64 then i ← 32
    EA ← EA + 1
    n ← n - 1

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

Special Registers Altered:

None

Store String Word Indexed  X-form

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>661</th>
<th>/</th>
</tr>
</thead>
</table>

stswx RS,RA,RB

if RA = 0 then b ← 0
else           b ← (RA)

EA ← b + (RB)
n ← XER57:63
r ← RS - 1
i ← 32

do while n > 0
    if i = 32 then r ← r + 1 (mod 32)
    MEM(EA, 1) ← GPR(r)i:i+7
    i ← i + 8
    if i = 64 then i ← 32
    EA ← EA + 1
    n ← n - 1

Let the effective address (EA) be the sum (RA|0)+(RB). Let n = XER57:63; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data.

If n>0, n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

If n=0, no bytes are stored.

Special Registers Altered:

None
3.3.8 Other Fixed-Point Instructions

The remainder of the fixed-point instructions use the contents of the General Purpose Registers (GPRs) as source operands, and place results into GPRs, into the Fixed-Point Exception Register (XER), and into Condition Register fields. In addition, the Trap instructions test the contents of a GPR or XER bit, invoking the system trap handler if the result of the specified test is true.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The X-form and XO-form instructions with \( R_c = 1 \), and the D-form instructions \textit{addic}. \textit{andi}. and \textit{andis}. set the first three bits of CR Field 0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to zero. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

Unless otherwise noted and when appropriate, when CR Field 0 and the XER are set they reflect the value placed into the target register.

<table>
<thead>
<tr>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions with the OE bit set or that set CA may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.</td>
</tr>
</tbody>
</table>
3.3.9 Fixed-Point Arithmetic Instructions

The XO-form Arithmetic instructions with Rc=1, and the D-form Arithmetic instruction addic., set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions”. addic, addic., subfic, addc, subfc, adde, subfe, addme, subfme, addze, and subfze always set CA, to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode. The XO-form Arithmetic instructions set SO and OV when OE=1 to reflect overflow of the result. Except for the Multiply Low and Divide instructions, the setting of these bits is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For XO-form Multiply Low and Divide instructions, the setting of these bits is mode-independent, and reflects overflow of the 64-bit result for mulld, divd, divide, divdu and divideu, and overflow of the low-order 32-bit result for mullw, divw, divwe, divwu, and divweu.

Extended mnemonics for addition and subtraction

Several extended mnemonics are provided that use the Add Immediate and Add Immediate Shifted instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

The Power ISA supplies Subtract From instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more “normal” order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate Add and Subtract From instructions.

See Appendix E for additional extended mnemonics.

Add Immediate
addi RT,RA,SI

D-form

| 14 | 5 | 11 | 16 | 31 |
---|---|---|---|---|
0  | 6 | RA | SI |

if RA = 0 then RT ← EXTS(SI)
else RT ← (RA) + EXTS(SI)

The sum (RA0) + SI is placed into register RT.

Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for Add Immediate:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>li Rx,value</td>
<td>addi Rx,0,value</td>
</tr>
<tr>
<td>la Rx,disp(Ry)</td>
<td>addi Rx,Ry,disp</td>
</tr>
<tr>
<td>subi Rx,Ry,value</td>
<td>addi Rx,Ry,-value</td>
</tr>
</tbody>
</table>

Programming Note
addi, addis, add, and subf are the preferred instructions for addition and subtraction, because they set few status bits.

Notice that addi and addis use the value 0, not the contents of GPR 0, if RA=0.

Add Immediate Shifted
addis RT,RA,SI

D-form

| 15 | 6 | 11 | 16 | 31 |
---|---|---|---|---|
0  | 5 | RA | SI |

if RA = 0 then RT ← EXTS(SI || 160)
else RT ← (RA) + EXTS(SI || 160)

The sum (RA0) + (SI || 0x0000) is placed into register RT.

Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for Add Immediate Shifted:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>lis Rx,value</td>
<td>addis Rx,0,value</td>
</tr>
<tr>
<td>subis Rx,Ry,value</td>
<td>addis Rx,Ry,-value</td>
</tr>
</tbody>
</table>
### Add

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonics</th>
<th>Form</th>
<th>Special Registers Altered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>add</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=0)</td>
</tr>
<tr>
<td></td>
<td>add.</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=1)</td>
</tr>
<tr>
<td></td>
<td>addo</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=0)</td>
</tr>
<tr>
<td></td>
<td>addo.</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=1)</td>
</tr>
</tbody>
</table>

**Example:**

\[
RT \leftarrow (RA) + (RB)
\]

The sum \((RA) + (RB)\) is placed into register RT.

**Special Registers Altered:**
- CR0 (if \(Rc=1\))
- SO OV (if \(OE=1\))

### Subtract From

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonics</th>
<th>Form</th>
<th>Special Registers Altered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract From</td>
<td>subf</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=0)</td>
</tr>
<tr>
<td></td>
<td>subf.</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=1)</td>
</tr>
<tr>
<td></td>
<td>subfo</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=0)</td>
</tr>
<tr>
<td></td>
<td>subfo.</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=1)</td>
</tr>
</tbody>
</table>

**Example:**

\[
RT \leftarrow \neg (RA) + (RB) + 1
\]

The sum \(\neg (RA) + (RB) + 1\) is placed into register RT.

**Special Registers Altered:**
- CR0 (if \(Rc=1\))
- SO OV (if \(OE=1\))

### Add Immediate Carrying

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonics</th>
<th>Form</th>
<th>Special Registers Altered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Immediate Carrying</td>
<td>addic</td>
<td>RT,RA,SI</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

\[
RT \leftarrow (RA) + \text{EXTS}(SI)
\]

The sum \((RA) + \text{SI}\) is placed into register RT.

**Special Registers Altered:**
- CA

### Add Immediate Carrying and Record

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonics</th>
<th>Form</th>
<th>Special Registers Altered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Immediate Carrying and Record</td>
<td>addic.</td>
<td>RT,RA,SI</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

\[
RT \leftarrow (RA) + \text{EXTS}(SI)
\]

The sum \((RA) + \text{SI}\) is placed into register RT.

**Special Registers Altered:**
- CR0 CA

### Extended Mnemonics:

- **Example of extended mnemonics for Add Immediate Carrying:**
  - **Extended:**
    - subic Rx,Ry,value
  - **Equivalent to:**
    - addic Rx,Ry,−value

- **Example of extended mnemonics for Add Immediate Carrying and Record:**
  - **Extended:**
    - subic. Rx,Ry,value
  - **Equivalent to:**
    - addic. Rx,Ry,−value
**Subtract From Immediate Carrying**

*D-form*

\[
\text{subfic} \quad \text{RT}, \text{RA}, \text{SI}
\]

\[
\begin{array}{c|c|c|c|c|c}
8 & 6 & 11 & 16 & 31 \\
0 & 6 & 11 & 16 & 31 \\
\end{array}
\]

\[\text{RT} \leftarrow \neg(\text{RA}) + \text{EXTS(SI)} + 1\]

The sum \(\neg(\text{RA}) + \text{SI} + 1\) is placed into register RT.

**Special Registers Altered:**
- CA

---

**Add Carrying**

*XO-form*

\[
\text{addc} \quad \text{RT}, \text{RA}, \text{RB} \\
\text{addc}. \quad \text{RT}, \text{RA}, \text{RB} \\
\text{addco} \quad \text{RT}, \text{RA}, \text{RB} \\
\text{addco}. \quad \text{RT}, \text{RA}, \text{RB}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
31 & 8 & 6 & 11 & 16 & 21 & 22 & 31 \\
0 & 6 & 11 & 16 & 21 & 22 & 31 \\
\end{array}
\]

\[\text{RT} \leftarrow (\text{RA}) + (\text{RB})\]

The sum \((\text{RA}) + (\text{RB})\) is placed into register RT.

**Special Registers Altered:**
- CA
- CR0 (if Rc=1)
- SO OV (if OE=1)

---

**Subtract From Carrying**

*XO-form*

\[
\text{subfc} \quad \text{RT}, \text{RA}, \text{RB} \\
\text{subfc}. \quad \text{RT}, \text{RA}, \text{RB} \\
\text{subfco} \quad \text{RT}, \text{RA}, \text{RB} \\
\text{subfco}. \quad \text{RT}, \text{RA}, \text{RB}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
31 & 8 & 6 & 11 & 16 & 21 & 22 & 31 \\
0 & 6 & 11 & 16 & 21 & 22 & 31 \\
\end{array}
\]

\[\text{RT} \leftarrow \neg(\text{RA}) + (\text{RB}) + 1\]

The sum \(\neg(\text{RA}) + (\text{RB}) + 1\) is placed into register RT.

**Special Registers Altered:**
- CA
- CR0 (if Rc=1)
- SO OV (if OE=1)

**Extended Mnemonics:**

Example of extended mnemonics for **Subtract From Carrying**:

\[
\text{Extended:} \quad \text{Equivalent to:}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\text{subc} \quad \text{Rx}, \text{Ry}, \text{Rz} & \text{subfc} \quad \text{Rx}, \text{Rz}, \text{Ry}
\end{array}
\]
### Add Extended XO-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA, RB</th>
<th>OE=0 Rc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addde</code></td>
<td>RT,RA, RB</td>
<td>OE=0 Rc=0</td>
</tr>
<tr>
<td><code>addde</code></td>
<td>RT,RA, RB</td>
<td>OE=1 Rc=0</td>
</tr>
<tr>
<td><code>adddeo</code></td>
<td>RT,RA, RB</td>
<td>OE=1 Rc=1</td>
</tr>
<tr>
<td><code>adddeo</code></td>
<td>RT,RA, RB</td>
<td>OE=1 Rc=1</td>
</tr>
</tbody>
</table>

The sum (RA) + (RB) + CA is placed into register RT.

### Subtract From Extended XO-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA, RB</th>
<th>OE=0 Rc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subfe</code></td>
<td>RT,RA, RB</td>
<td>OE=0 Rc=0</td>
</tr>
<tr>
<td><code>subfe</code></td>
<td>RT,RA, RB</td>
<td>OE=1 Rc=0</td>
</tr>
<tr>
<td><code>subfeo</code></td>
<td>RT,RA, RB</td>
<td>OE=1 Rc=1</td>
</tr>
<tr>
<td><code>subfeo</code></td>
<td>RT,RA, RB</td>
<td>OE=1 Rc=1</td>
</tr>
</tbody>
</table>

The sum ¬(RA) + (RB) + CA is placed into register RT.

### Add to Minus One Extended XO-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA</th>
<th>OE=0 Rc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addme</code></td>
<td>RT,RA</td>
<td>OE=0 Rc=0</td>
</tr>
<tr>
<td><code>addme</code></td>
<td>RT,RA</td>
<td>OE=1 Rc=0</td>
</tr>
<tr>
<td><code>addmeo</code></td>
<td>RT,RA</td>
<td>OE=1 Rc=1</td>
</tr>
<tr>
<td><code>addmeo</code></td>
<td>RT,RA</td>
<td>OE=1 Rc=1</td>
</tr>
</tbody>
</table>

The sum (RA) + CA - 1 is placed into register RT.

### Subtract From Minus One Extended XO-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA</th>
<th>OE=0 Rc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subfme</code></td>
<td>RT,RA</td>
<td>OE=0 Rc=0</td>
</tr>
<tr>
<td><code>subfme</code></td>
<td>RT,RA</td>
<td>OE=1 Rc=0</td>
</tr>
<tr>
<td><code>subfmeo</code></td>
<td>RT,RA</td>
<td>OE=1 Rc=1</td>
</tr>
<tr>
<td><code>subfmeo</code></td>
<td>RT,RA</td>
<td>OE=1 Rc=1</td>
</tr>
</tbody>
</table>

The sum ¬(RA) + CA - 1 is placed into register RT.
### Add to Zero Extended  
**XO-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RA</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>addze</td>
<td>RT, RA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addze.</td>
<td>RT, RA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addzeo</td>
<td>RT, RA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>addzeo.</td>
<td>RT, RA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum \((RA) + CA\) is placed into register RT.

**Special Registers Altered:**
- CA
- CR0 (if Rc=1)
- SO OV (if OE=1)

### Subtract From Zero Extended  
**XO-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RA</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfze</td>
<td>RT, RA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subfze.</td>
<td>RT, RA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>subfzeo</td>
<td>RT, RA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subfzeo.</td>
<td>RT, RA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum \(\neg(RA) + CA\) is placed into register RT.

**Special Registers Altered:**
- CA
- CR0 (if Rc=1)
- SO OV (if OE=1)

---

### Negate  
**XO-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RA</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg</td>
<td>RT, RA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>neg.</td>
<td>RT, RA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nego</td>
<td>RT, RA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nego.</td>
<td>RT, RA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum \(\neg(RA) + 1\) is placed into register RT.

If the processor is in 64-bit mode and register RA contains the most negative 64-bit number \((0x8000_0000_0000_0000)\), the result is the most negative number and, if OE=1, OV is set to 1. Similarly, if the processor is in 32-bit mode and \((RA)_{32:63}\) contain the most negative 32-bit number \((0x8000_0000)\), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV is set to 1.

**Special Registers Altered:**
- CR0 (if Rc=1)
- SO OV (if OE=1)
### Multiply Low Immediate  

**D-form**

\[
mulli \quad RT, RA, SI
\]

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RT</td>
<td>RA</td>
<td>SI</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{prod}_{0:127} \leftarrow (\text{RA}) \times \text{EXTS}[\text{SI}]\]

\[\text{RT} \leftarrow \text{prod}_{64:127}\]

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the SI field. The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

None

### Multiply Low Word  

**XO-form**

\[
mullw \quad \text{RT}, RA, RB (\text{OE}=0 \text{ Rc}=0)
mullw. \quad \text{RT}, RA, RB (\text{OE}=0 \text{ Rc}=1)
mullwo \quad \text{RT}, RA, RB (\text{OE}=1 \text{ Rc}=0)
mullwo. \quad \text{RT}, RA, RB (\text{OE}=1 \text{ Rc}=1)
\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>OE</td>
<td>235</td>
<td>Rc</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{RT}_{32:63} \leftarrow (\text{RA}_{32:63} \times \text{RB}_{32:63})\]

The 32-bit operands are the low-order 32 bits of RA and of RB. The 64-bit product of the operands is placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

\[\text{CR0} \quad (\text{if Rc}=1)\]
\[\text{SO OV} \quad (\text{if OE}=1)\]

#### Programming Note

For \textit{mulli} and \textit{mullw}, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For \textit{mulli} and \textit{mulld}, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers.

For \textit{mulli} and \textit{mullw}, the low-order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

### Multiply High Word  

**XO-form**

\[
mulhw \quad \text{RT}, RA, RB (\text{Rc}=0)
mulhw. \quad \text{RT}, RA, RB (\text{Rc}=1)
\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>75</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>OE</td>
<td>235</td>
<td>/</td>
<td>22</td>
<td>Rc</td>
</tr>
</tbody>
</table>

\[\text{prod}_{0:63} \leftarrow (\text{RA}_{32:63} \times \text{RB}_{32:63})\]

\[\text{RT}_{32:63} \leftarrow \text{prod}_{0:63}\]

\[\text{RT}_{0:31} \leftarrow \text{undefined}\]

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the operands are placed into RT\text{.}_{32:63}. The contents of RT\text{.}_{0:31} are undefined.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

\[\text{CR0} \quad (\text{bits 0:2 undefined in 64-bit mode}) \quad (\text{if Rc}=1)\]

### Multiply High Word Unsigned  

**XO-form**

\[
mulhwu \quad \text{RT}, RA, RB (\text{Rc}=0)
mulhwu. \quad \text{RT}, RA, RB (\text{Rc}=1)
\]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>11</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>OE</td>
<td>235</td>
<td>/</td>
<td>22</td>
<td>Rc</td>
</tr>
</tbody>
</table>

\[\text{prod}_{0:63} \leftarrow (\text{RA}_{32:63} \times \text{RB}_{32:63})\]

\[\text{RT}_{32:63} \leftarrow \text{prod}_{0:63}\]

\[\text{RT}_{0:31} \leftarrow \text{undefined}\]

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the operands are placed into RT\text{.}_{32:63}. The contents of RT\text{.}_{0:31} are undefined.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

**Special Registers Altered:**

\[\text{CR0} \quad (\text{bits 0:2 undefined in 64-bit mode}) \quad (\text{if Rc}=1)\]
Divide Word

**XO-form**

- `divw` RT,RA,RB (OE=0 Rc=0)
- `divw` RT,RA,RB (OE=0 Rc=1)
- `divwo` RT,RA,RB (OE=1 Rc=0)
- `divwo` RT,RA,RB (OE=1 Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
</table>
| dividend_{0:31} &lt; (RA)_{32:63}  
divisor_{0:31} &lt; (RB)_{32:63}  
RT_{32:63} &lt; dividend + divisor  
RT_{0:31} &lt; undefined |

The 32-bit dividend is (RA)_{32:63}. The 32-bit divisor is (RB)_{32:63}. The 32-bit quotient is placed into RT_{32:63}. The contents of RT_{0:31} are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
dividend = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < r \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
0x8000_0000 \div -1  
<\text{anything}> \div 0
\]

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EO bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

**Special Registers Altered:**

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO OV (if OE=1)

**Programming Note**

The 32-bit signed remainder of dividing (RA)_{32:63} by (RB)_{32:63} can be computed as follows, except in the case that (RA)_{32:63} = -2^{31} and (RB)_{32:63} = -1.

\[
\text{divw} \quad \text{RT},\text{RA},\text{RB} \quad \# \ RT = \text{quotient}  
\text{mullw} \quad \text{RT},\text{RT},\text{RB} \quad \# \ RT = \text{quotient} \times \text{divisor}  
\text{subf} \quad \text{RT},\text{RT},\text{RA} \quad \# \ RT = \text{remainder}
\]

Divide Word Unsigned

**XO-form**

- `divwu` RT,RA,RB (OE=0 Rc=0)
- `divwu` RT,RA,RB (OE=0 Rc=1)
- `divwu` RT,RA,RB (OE=1 Rc=0)
- `divwu` RT,RA,RB (OE=1 Rc=1)

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
</table>
| dividend_{0:31} &lt; (RA)_{32:63}  
divisor_{0:31} &lt; (RB)_{32:63}  
RT_{32:63} &lt; dividend + divisor  
RT_{0:31} &lt; undefined |

The 32-bit dividend is (RA)_{32:63}. The 32-bit divisor is (RB)_{32:63}. The 32-bit quotient is placed into RT_{32:63}. The contents of RT_{0:31} are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

\[
dividend = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < \text{divisor}\).

If an attempt is made to perform the division

\[
<\text{anything}> \div 0
\]

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EO bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

**Special Registers Altered:**

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO OV (if OE=1)

**Programming Note**

The 32-bit unsigned remainder of dividing (RA)_{32:63} by (RB)_{32:63} can be computed as follows.

\[
\text{divwu} \quad \text{RT},\text{RA},\text{RB} \quad \# \ RT = \text{quotient}  
\text{mullw} \quad \text{RT},\text{RT},\text{RB} \quad \# \ RT = \text{quotient} \times \text{divisor}  
\text{subf} \quad \text{RT},\text{RT},\text{RA} \quad \# \ RT = \text{remainder}
\]
**Divide Word Extended**

<table>
<thead>
<tr>
<th>Operation</th>
<th>RT, RA, RB</th>
<th>OE=0 Rc=0</th>
<th>OE=0 Rc=1</th>
<th>OE=1 Rc=0</th>
<th>OE=1 Rc=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>divwe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divwe.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divweo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divweuo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Category: Server]
[Category: Embedded.Phased-In]

**Dividend**

\[
\text{dividend}_{0:63} \leftarrow (RA)_{32:63} \ || \ 320
\]

**Divisor**

\[
\text{divisor}_{0:31} \leftarrow (RB)_{32:63}
\]

**Quotient**

\[
RT_{32:63} \leftarrow \text{dividend} + \text{divisor}
\]

\[
RT_{0:31} \leftarrow \text{undefined}
\]

The 64-bit dividend is \((RA)_{32:63} \ || \ 320\). The 32-bit divisor is \((RB)_{32:63}\). If the quotient can be represented in 32 bits, it is placed into \(RT_{32:63}\). The contents of \(RT_{0:31}\) are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < r \leq 0\) if the dividend is negative.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division

\[
\langle\text{anything} \rangle \div 0
\]

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

**Special Registers Altered:**

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO OV (if OE=1)

---

**Divide Word Extended Unsigned**

<table>
<thead>
<tr>
<th>Operation</th>
<th>RT, RA, RB</th>
<th>OE=0 Rc=0</th>
<th>OE=0 Rc=1</th>
<th>OE=1 Rc=0</th>
<th>OE=1 Rc=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>divweu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divweu.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divweuo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divweuuo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Category: Server]
[Category: Embedded.Phased-In]

**Dividend**

\[
\text{dividend}_{0:63} \leftarrow (RA)_{32:63} \ || \ 320
\]

**Divisor**

\[
\text{divisor}_{0:31} \leftarrow (RB)_{32:63}
\]

**Quotient**

\[
RT_{32:63} \leftarrow \text{dividend} + \text{divisor}
\]

\[
RT_{0:31} \leftarrow \text{undefined}
\]

The 64-bit dividend is \((RA)_{32:63} \ || \ 320\). The 32-bit divisor is \((RB)_{32:63}\). If the quotient can be represented in 32 bits, it is placed into \(RT_{32:63}\). The contents of \(RT_{0:31}\) are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < \text{divisor}\).

If (RA) \(\geq\) (RB), or if an attempt is made to perform the division

\[
\langle\text{anything} \rangle \div 0
\]

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

**Special Registers Altered:**

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO OV (if OE=1)
Programming Note

Unsigned long division of a 64-bit dividend contained in two 32-bit registers by a 32-bit divisor can be computed as follows. The algorithm is shown first, followed by Assembler code that implements the algorithm. The dividend is \( \text{Dh} \| \text{Dl} \), the divisor is \( \text{Dv} \), and the quotient and remainder are \( Q \) and \( R \) respectively, where these variables and all intermediate variables represent unsigned 32-bit integers. It is assumed that \( \text{Dv} > \text{Dh} \), and that assigning a value to an intermediate variable assigns the low-order 32 bits of the value and ignores any higher-order bits of the value. (In both the algorithm and the Assembler code, “r1” and “r2” refer to “remainder 1” and “remainder 2”, rather than to GPRs 1 and 2.)

Algorithm:

3. \( q_1 \leftarrow \text{divweu} \text{Dh, Dv} \) 
4. \( r_1 \leftarrow -(q_1 \times \text{Dv}) \) \# remainder of step 1
   divide operation (see Note 1)
5. \( q_2 \leftarrow \text{divwu} \text{Dl, Dv} \)
6. \( r_2 \leftarrow \text{Dl} - (q_2 \times \text{Dv}) \) \# remainder of step 2
   divide operation
7. \( Q \leftarrow q_1 + q_2 \)
8. \( R \leftarrow r_1 + r_2 \)
9. if \( (R < r_2) \) \| \( (R \geq \text{Dv}) \) then \# (see Note 2)
   \( Q \leftarrow Q + 1 \) \# increment quotient
   \( R \leftarrow R - \text{Dv} \) \# decrement remainder

Assembler Code:

\[
\begin{align*}
# \text{Dh in r4, Dl in r5} \\
# \text{Dv in r6} \\
\text{divweu} \ r3, r4, r6 & \quad # q1 \\
\text{divwu} \ r7, r5, r6 & \quad # q2 \\
\text{mullw} \ r8, r3, r6 & \quad # -r1 = q1 \times \text{Dv} \\
\text{mullw} \ r0, r7, r6 & \quad # q2 \times \text{Dv} \\
\text{subf} \ r10, r0, r5 & \quad # r2 = \text{Dl} - (q2 \times \text{Dv}) \\
\text{add} \ r3, r3, r7 & \quad # Q = q1 + q2 \\
\text{subf} \ r4, r8, r10 & \quad # R = r1 + r2 \\
\text{cmplw} \ r4, r10 & \quad # R < r2 \quad ? \\
\text{blt} \ ^*+12 & \quad # \text{must adjust Q and R if yes} \\
\text{cmplw} \ r4, r6 & \quad # R \geq \text{Dv} \quad ? \\
\text{blt} \ ^*+12 & \quad # \text{must adjust Q and R if yes} \\
\text{addi} \ r3, r3, 1 & \quad # Q = Q + 1 \\
\text{subf} \ r4, r6, r4 & \quad # R = R - \text{Dv} \\
# \text{Quotient in r3} \\
# \text{Remainder in r4}
\end{align*}
\]

Notes:

1. The remainder is \( \text{Dh} \|^{320} (q_1 \times \text{Dv}) \). Because the remainder must be less than \( \text{Dv} \) and \( \text{Dv} < 2^{32} \), the remainder is representable in 32 bits. Because the low-order 32 bits of \( \text{Dh} \|^{320} \) are 0s, the remainder is therefore equal to the low-order 32 bits of \( -(q_1 \times \text{Dv}) \). Thus assigning \( -(q_1 \times \text{Dv}) \) to \( r1 \) yields the correct remainder.
2. \( R \) is less than \( r2 \) (and also less than \( r1 \)) if and only if the addition at step 6 carried out of 32 bits — i.e., if and only if the correct sum could not be represented in 32 bits — in which case the correct sum is necessarily greater than \( \text{Dv} \).
### 3.3.9.1 64-bit Fixed-Point Arithmetic Instructions [Category: 64-Bit]

#### Multiply Low Doubleword

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>64-bit Instruction</th>
<th>OE</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulld</code></td>
<td>RT, RA, RB</td>
<td>(OE=0 RC=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mulld</code></td>
<td>RT, RA, RB</td>
<td>(OE=0 RC=1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mulldo</code></td>
<td>RT, RA, RB</td>
<td>(OE=1 RC=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mulldo</code></td>
<td>RT, RA, RB</td>
<td>(OE=1 RC=1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

product:
```
prod0:127
```

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:
- CR0 (if RC=1)
- SO OV (if OE=1)

#### Programming Note

The XO-form `Multiply` instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

#### Multiply High Doubleword

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>64-bit Instruction</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulhd</code></td>
<td>RT, RA, RB</td>
<td>(RC=0)</td>
<td></td>
</tr>
<tr>
<td><code>mulhd</code></td>
<td>RT, RA, RB</td>
<td>(RC=1)</td>
<td></td>
</tr>
</tbody>
</table>

product:
```
prod0:127 ← (RA) × (RB)
```

RT ← `prod0:63`

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:
- CR0 (if RC=1)

#### Multiply High Doubleword Unsigned

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>64-bit Instruction</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulhdu</code></td>
<td>RT, RA, RB</td>
<td>(RC=0)</td>
<td></td>
</tr>
<tr>
<td><code>mulhdu</code></td>
<td>RT, RA, RB</td>
<td>(RC=1)</td>
<td></td>
</tr>
</tbody>
</table>

product:
```
prod0:127 ← (RA) × (RB)
```

RT ← `prod0:63`

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as unsigned integers, except that if RC=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:
- CR0 (if RC=1)
### Divide Doubleword

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>divd</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=0)</td>
</tr>
<tr>
<td>divd.</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=1)</td>
</tr>
<tr>
<td>divdo</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=0)</td>
</tr>
<tr>
<td>divdo.</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=1)</td>
</tr>
</tbody>
</table>

Dividend0:63 \( \leftarrow (RA) \)

Divisor0:63 \( \leftarrow (RB) \)

The 64-bit dividend is \((RA)\). The 64-bit divisor is \((RB)\). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
dividend = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < r \leq 0\) if the dividend is negative.

If an attempt is made to perform any of the divisions

\[
0x8000_0000_0000_0000 \div \text{<anything>}
\]

then the contents of register RT are undefined as are \((\text{if Rc=1})\) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

**Special Registers Altered:**

- CR0 (if Rc=1)
- SO OV (if OE=1)

### Programming Note

The 64-bit signed remainder of dividing \((RA)\) by \((RB)\) can be computed as follows, except in the case that \((RA) = -2^{63}\) and \((RB) = -1\).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>divd</td>
<td>RT,RA,RB</td>
<td># RT = quotient</td>
</tr>
<tr>
<td>mulld</td>
<td>RT,RT,RB</td>
<td># RT = quotientxdivisor</td>
</tr>
<tr>
<td>subf</td>
<td>RT,RT,RA</td>
<td># RT = remainder</td>
</tr>
</tbody>
</table>

### Divide Doubleword Unsigned

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>divdu</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=0)</td>
</tr>
<tr>
<td>divdu.</td>
<td>RT,RA,RB</td>
<td>(OE=0 Rc=1)</td>
</tr>
<tr>
<td>divduo</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=0)</td>
</tr>
<tr>
<td>divduo.</td>
<td>RT,RA,RB</td>
<td>(OE=1 Rc=1)</td>
</tr>
</tbody>
</table>

Dividend0:63 \( \leftarrow (RA) \)

Divisor0:63 \( \leftarrow (RB) \)

The 64-bit dividend is \((RA)\). The 64-bit divisor is \((RB)\). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

\[
dividend = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < \text{divisor}\).

If an attempt is made to perform the division

\[
\text{<anything>} \div 0
\]

then the contents of register RT are undefined as are \((\text{if Rc=1})\) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

**Special Registers Altered:**

- CR0 (if Rc=1)
- SO OV (if OE=1)

### Programming Note

The 64-bit unsigned remainder of dividing \((RA)\) by \((RB)\) can be computed as follows.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>divdu</td>
<td>RT,RA,RB</td>
<td># RT = quotient</td>
</tr>
<tr>
<td>mulld</td>
<td>RT,RT,RB</td>
<td># RT = quotientxdivisor</td>
</tr>
<tr>
<td>subf</td>
<td>RT,RT,RA</td>
<td># RT = remainder</td>
</tr>
</tbody>
</table>
Divide Doubleword Extended  
XO-form

divde RT,RA,RB (OE=0 Rc=0)
divde. RT,RA,RB (OE=0 Rc=1)
divdeo RT,RA,RB (OE=1 Rc=0)
divdeo. RT,RA,RB (OE=1 Rc=1)

[Category: Server]
[Category: Embedded.Phased-In]

Divide Doubleword Extended Unsigned  
XO-form

divdeu RT,RA,RB (OE=0 Rc=0)
divdeu. RT,RA,RB (OE=0 Rc=1)
divdeuo RT,RA,RB (OE=1 Rc=0)
divdeuo. RT,RA,RB (OE=1 Rc=1)

[Category: Server]
[Category: Embedded.Phased-In]

The 128-bit dividend is (RA) \| 640. The 64-bit divisor is (RB). If the quotient can be represented in 64 bits, it is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < |\text{divisor}|\) if the dividend is nonnegative, and \(-|\text{divisor}| < r \leq 0\) if the dividend is negative.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division

\(<\text{anything}> \div 0\)

then the contents of register RT are undefined as are (if \(\text{Rc}=1\)) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

<table>
<thead>
<tr>
<th>CR0</th>
<th>(if (\text{Rc}=1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO</td>
<td>(if OE=1)</td>
</tr>
</tbody>
</table>

Programming Note

Unsigned long division of a 128-bit dividend contained in two 64-bit registers by a 64-bit divisor can be accomplished using the technique described in the Programming Note with the \texttt{divdeu} instruction description: \texttt{divdeu} would be used instead of \texttt{divweu} (and \texttt{cmpld} instead of \texttt{cmplw}, etc.).
3.3.10 Fixed-Point Compare Instructions

The fixed-point Compare instructions compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of register RB. The comparison is signed for cmpi and cmp, and unsigned for cmpli and cmpl.

The L field controls whether the operands are treated as 64-bit or 32-bit quantities, as follows:

<table>
<thead>
<tr>
<th>L</th>
<th>Operand length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32-bit operands</td>
</tr>
<tr>
<td>1</td>
<td>64-bit operands</td>
</tr>
</tbody>
</table>

L=1 is part of Category: 64-Bit.

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

The Compare instructions set one bit in the leftmost three bits of the designated CR field to 1, and the other two to 0. XERSO is copied to bit 3 of the designated CR field.

The CR field is set as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>(RA) &lt; SI or (RB) (signed comparison)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RA) &lt; Ui or (RB) (unsigned comparison)</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>(RA) &gt; SI or (RB) (signed comparison)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RA) &gt; Ui or (RB) (unsigned comparison)</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>(RA) = SI, UI, or (RB)</td>
</tr>
<tr>
<td>3</td>
<td>SO</td>
<td>Summary Overflow from the XER</td>
</tr>
</tbody>
</table>

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the Compare instructions. See Appendix E for additional extended mnemonics.

### Compare Immediate

<table>
<thead>
<tr>
<th>Compare</th>
<th>BF,L,RA,SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpli</td>
<td></td>
</tr>
<tr>
<td>cmpli</td>
<td></td>
</tr>
</tbody>
</table>

#### D-form

- **cmp**\(BF, L, RA, SI\)
- **cmp**\(BF, L, RA, SI\)

#### X-form

- **cmp**\(BF, L, RA, RB\)
- **cmp**\(BF, L, RA, RB\)

**Extended Mnemonics:**

**Example of extended mnemonics for Compare Immediate:**

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpli Rx,value</td>
<td>cmpl 0,Rx,value</td>
</tr>
<tr>
<td>cmpli cr3,Rx,value</td>
<td>cmpl 3,Rx,value</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**

- CR field BF

**Extended Mnemonics:**

**Example of extended mnemonics for Compare:**

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpd Rx,Ry</td>
<td>cmp 0,Rx,Ry</td>
</tr>
<tr>
<td>cmplw cr3,Rx,Ry</td>
<td>cmpl 3,Rx,Ry</td>
</tr>
</tbody>
</table>
### Compare Logical Immediate

**D-form**

<table>
<thead>
<tr>
<th>cmpli</th>
<th>BF,L,RA,UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
</tr>
<tr>
<td>10</td>
<td>RA</td>
</tr>
<tr>
<td>16</td>
<td>UI</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if L = 0 then $a \leftarrow 32:0 || (RA)_{32:63}$
else $a \leftarrow (RA)$

if $a <_{u} (480 || UI)$ then $c \leftarrow 0b100$
else if $a >_{u} (480 || UI)$ then $c \leftarrow 0b010$
else $c \leftarrow 0b001$

$CR_{BF+32:4} \times BF+35 \leftarrow c || XERSO$

The contents of register RA ((RA)_{32:63} zero-extended to 64 bits if $L=0$) are compared with $480 || UI$, treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

**Special Registers Altered:**
CR field BF

**Extended Mnemonics:**

Examples of extended mnemonics for Compare Logical Immediate:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpldi Nx.value</td>
<td>cmpl 0,1,Nx.value</td>
</tr>
<tr>
<td>cmplwi cr3,Nx.value</td>
<td>cmpl 3,0,Nx.value</td>
</tr>
</tbody>
</table>

**Compare Logical**

**X-form**

<table>
<thead>
<tr>
<th>cmpl</th>
<th>BF,L,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>L</td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>16</td>
<td>RB</td>
</tr>
<tr>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if L = 0 then $a \leftarrow 32:0 || (RA)_{32:63}$
$b \leftarrow 32:0 || (RB)_{32:63}$
else $a \leftarrow (RA)$
$b \leftarrow (RB)$

if $a <_{u} b$ then $c \leftarrow 0b100$
else if $a >_{u} b$ then $c \leftarrow 0b010$
else $c \leftarrow 0b001$

$CR_{BF+32:4} \times BF+35 \leftarrow c || XERSO$

The contents of register RA ((RA)_{32:63} if $L=0$) are compared with the contents of register RB ((RB)_{32:63} if $L=0$), treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

**Special Registers Altered:**
CR field BF

**Extended Mnemonics:**

Examples of extended mnemonics for Compare Logical:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpld Rx,Ry</td>
<td>cmpl 0,1,Rx,Ry</td>
</tr>
<tr>
<td>cmplw cr3,Rx,Ry</td>
<td>cmpl 3,0,Rx,Ry</td>
</tr>
</tbody>
</table>
3.3.11 Fixed-Point Trap Instructions

The Trap instructions are provided to test for a specified set of conditions. If any of the conditions tested by a Trap instruction are met, the system trap handler is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of register RA are compared with either the sign-extended value of the SI field or the contents of register RB, depending on the Trap instruction. For tdi and td, the entire contents of RA (and RB) participate in the comparison; for twi and tw, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

This comparison results in five conditions which are ANDed with TO. If the result is not 0 the system trap handler is invoked. These conditions are as follows.

<table>
<thead>
<tr>
<th>TO Bit</th>
<th>ANDed with Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less Than, using signed comparison</td>
</tr>
<tr>
<td>1</td>
<td>Greater Than, using signed comparison</td>
</tr>
<tr>
<td>2</td>
<td>Equal</td>
</tr>
<tr>
<td>3</td>
<td>Less Than, using unsigned comparison</td>
</tr>
<tr>
<td>4</td>
<td>Greater Than, using unsigned comparison</td>
</tr>
</tbody>
</table>

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the Trap instructions. See Appendix E for additional extended mnemonics.

<table>
<thead>
<tr>
<th>Trap Word Immediate</th>
<th>D-form</th>
<th>Trap Word X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>twi TO,RA,SI</td>
<td></td>
<td>tw TO,RA,RB</td>
</tr>
<tr>
<td>0 3 6 11 16 SI 31</td>
<td></td>
<td>0 31 6 11 16 21 31</td>
</tr>
<tr>
<td>a ← EXTS((RA)32:63)</td>
<td></td>
<td>a ← EXTS((RA)32:63)</td>
</tr>
<tr>
<td>if (a &lt; EXTS(SI)) &amp; TO0 then TRAP</td>
<td>b ← EXTS((RB)32:63)</td>
<td></td>
</tr>
<tr>
<td>if (a &gt; EXTS(SI)) &amp; TO1 then TRAP</td>
<td>if (a &lt; b) &amp; TO0 then TRAP</td>
<td></td>
</tr>
<tr>
<td>if (a = EXTS(SI)) &amp; TO2 then TRAP</td>
<td>if (a &gt; b) &amp; TO0 then TRAP</td>
<td></td>
</tr>
<tr>
<td>if (a &lt;u EXTS(SI)) &amp; TO3 then TRAP</td>
<td>if (a = b) &amp; TO2 then TRAP</td>
<td></td>
</tr>
<tr>
<td>if (a &gt;u EXTS(SI)) &amp; TO4 then TRAP</td>
<td>if (a &lt;u b) &amp; TO3 then TRAP</td>
<td></td>
</tr>
</tbody>
</table>

The contents of RA32:63 are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered: None

Extended Mnemonics:

Examples of extended mnemonics for Trap Word Immediate:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>twg ti</td>
<td>twi 8,Rx,value</td>
</tr>
<tr>
<td>tw llei</td>
<td>twi 6,Rx,value</td>
</tr>
</tbody>
</table>

Extended: Equivalent to:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>tweq</td>
<td>tw 4,Rx,Ry</td>
</tr>
<tr>
<td>twlge</td>
<td>tw 5,Rx,Ry</td>
</tr>
<tr>
<td>trap</td>
<td>tw 31,0,0</td>
</tr>
</tbody>
</table>

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered: None

Extended Mnemonics:

Examples of extended mnemonics for Trap Word:
3.3.11.1 64-bit Fixed-Point Trap Instructions [Category: 64-Bit]

**Trap Doubleword Immediate**  
**D-form**

\[
\text{tdi TO,RA,SI} \quad \text{SI} = 2^b \times (RA) + a
\]

The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

**Special Registers Altered:**  
None

**Extended Mnemonics:**

Examples of extended mnemonics for **Trap Doubleword Immediate**:

- Extended: `tditi Rx,value`  
  Equivalent to: `tdi 16,Rx,value`
- Extended: `tdnei Rx,value`  
  Equivalent to: `tdi 24,Rx,value`

**Trap Doubleword**  
**X-form**

\[
\text{td TO,RA,RB} \quad \text{RB} = 2^c \times (RB) + b
\]

The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

**Special Registers Altered:**  
None

**Extended Mnemonics:**

Examples of extended mnemonics for **Trap Doubleword**:

- Extended: `tdge Rx,Ry`  
  Equivalent to: `td 12,Rx,Ry`

3.3.12 Fixed-Point Select [Category: Phased-In (sV2.06)]

**Integer Select**  
**A-form**

\[
isel RT,RA,RB,BC \quad \text{BC} = 2^d \times (RB) + c
\]

If the contents of bit BC+32 of the Condition Register are equal to 1, then the contents of register RA (or 0) are placed into register RT. Otherwise, the contents of register RB are placed into register RT.

**Special Registers Altered:**  
None

**Extended Mnemonics:**

Examples of extended mnemonics for **Integer Select**:

- Extended: `isellt Rx,Ry,Rz`  
  Equivalent to: `isel Rx,Ry,Rz,0`
- Extended: `iselgt Rx,Ry,Rz`  
  Equivalent to: `isel Rx,Ry,Rz,1`
- Extended: `iselleq Rx,Ry,Rz`  
  Equivalent to: `isel Rx,Ry,Rz,1`
3.3.13 Fixed-Point Logical Instructions

The Logical instructions perform bit-parallel operations on 64-bit operands.

The X-form Logical instructions with Rc=1, and the D-form Logical instructions andi. and andis., set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 65. The Logical instructions do not change the SO, OV, and CA bits in the XER.

Extended mnemonics for logical operations

Extended mnemonics are provided that generate two different types of “no-ops” (instructions that do nothing). The first type is the preferred form, which is optimized to minimize its use of the processor’s execution resources. This form is based on the OR Immediate instruction. The second type is the executed form, which is intended to consume the same amount of the processor’s execution resources as if it were not a no-op. This form is based on the XOR Immediate instruction. (There are also no-ops that have other uses, such as affecting program priority, for which extended mnemonics have not been defined.)

Extended mnemonics are provided that use the OR and NOR instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

See Appendix E, “Assembler Extended Mnemonics” on page 709 for additional extended mnemonics.

---

**Programming Note**

Warning: Some forms of no-op may have side effects such as affecting program priority. Programmers should use the preferred no-op unless the side effects of some other form of no-op are intended.

---

**AND Immediate**  
D-form

<table>
<thead>
<tr>
<th>andi.</th>
<th>RA,RS,UI</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>28</th>
<th>RS</th>
<th>31</th>
</tr>
</thead>
</table>

RA ← (RS) & (480 || UI)

The contents of register RS are ANDed with 480 || UI and the result is placed into register RA.

**Special Registers Altered:**

CR0

**AND Immediate Shifted**  
D-form

<table>
<thead>
<tr>
<th>andis.</th>
<th>RA,RS,UI</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>29</th>
<th>RS</th>
<th>31</th>
</tr>
</thead>
</table>

RA ← (RS) & (320 || 160)

The contents of register RS are ANDed with 320 || 160 and the result is placed into register RA.

**Special Registers Altered:**

CR0

**OR Immediate**  
D-form

<table>
<thead>
<tr>
<th>ori</th>
<th>RA,RS,UI</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>24</th>
<th>RS</th>
<th>31</th>
</tr>
</thead>
</table>

RA ← (RS) | (480 || UI)

The contents of register RS are ORed with 480 || UI and the result is placed into register RA.

The preferred “no-op” (an instruction that does nothing) is:

ori 0,0,0

**Special Registers Altered:**

None

**Extended Mnemonics:**

Example of extended mnemonics for OR Immediate:

Extended:  
Equivalent to:

- no-op  
  ori 0,0,0
**OR Immediate Shifted**

**D-form**

\[ \text{oris} \quad \text{RA,RS,UI} \]

\[ \begin{array}{cccccc}
25 & 6 & 11 & 16 & 31 \\
\hline
\end{array} \]

\[ \text{RA} \leftarrow (\text{RS}) \mid (320 \mid \text{UI} \mid 160) \]

The contents of register RS are ORed with \(320 \parallel \text{UI} \parallel 160\) and the result is placed into register RA.

**Special Registers Altered:**

None

---

**XOR Immediate**

**D-form**

\[ \text{xori} \quad \text{RA,RS,UI} \]

\[ \begin{array}{cccccc}
26 & 6 & 11 & 16 & 31 \\
\hline
\end{array} \]

\[ \text{RA} \leftarrow (\text{RS}) \oplus (480 \mid \text{UI}) \]

The contents of register RS are XORed with \(480 \parallel \text{UI}\) and the result is placed into register RA.

The executed form of a “no-op” (an instruction that does nothing, but consumes execution resources nevertheless) is:

\[ \text{xori} \ 0,0,0 \]

**Special Registers Altered:**

None

**Extended Mnemonics:**

Example of extended mnemonics for **XOR Immediate**:

\[ \text{Extended: } \quad \text{Equivalent to:} \]

\[ \text{xnop} \quad \text{xori} \ 0,0,0 \]

---

**Programming Note**

The executed form of no-op should be used only when the intent is to alter the timing of a program.
**AND**

\[
\text{and} \quad \text{RA,RS,RB} \quad (\text{Rc}=0) \\
\text{and.} \quad \text{RA,RS,RB} \quad (\text{Rc}=1)
\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & 21 & 28 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

RA \leftarrow (RS) \& (RB)

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

**X-form**

Some forms of **and** Rx, Rx, Rx may provide special functions, see Section 9.3 of Book III-S.

**Special Registers Altered:**

CR0 \hspace{1cm} (if Rc=1)

---

**OR**

\[
\text{or} \quad \text{RA,RS,RB} \quad (\text{Rc}=0) \\
\text{or.} \quad \text{RA,RS,RB} \quad (\text{Rc}=1)
\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & 21 & 28 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

RA \leftarrow (RS) \oplus (RB)

The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

**X-form**

Some forms of **or** Rx,Rx,Rx provide special functions. See Section 3.2, ""or" Instruction", in Book II.

**Special Registers Altered:**

CR0 \hspace{1cm} (if Rc=1)

---

**NAND**

\[
\text{nand} \quad \text{RA,RS,RB} \quad (\text{Rc}=0) \\
\text{nand.} \quad \text{RA,RS,RB} \quad (\text{Rc}=1)
\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & 21 & 28 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

RA \leftarrow \neg ((RS) \& (RB))

The contents of register RS are ANDed with the contents of register RB and the complemented result is placed into register RA.

**X-form**

**Special Registers Altered:**

CR0 \hspace{1cm} (if Rc=1)

---

**Programming Note**

**nand** or **nor** with RS=RB can be used to obtain the one's complement.
Chapter 3. Fixed-Point Facility

### NOR

**X-form**

nor    RA,RS,RB
nor.   RA,RS,RB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Equivalent**

eqv   RA,RS,RB
eqv.  RA,RS,RB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of register RS are ORed with the contents of register RB and the complemented result is placed into register RA.

**Special Registers Altered:**

CR0 (if Rc=1)

**Extended Mnemonics:**

Example of extended mnemonics for NOR:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>nor.</td>
<td>nor Rx, Ry</td>
</tr>
</tbody>
</table>

### AND with Complement

**X-form**

andc   RA,RS,RB
andc.  RA,RS,RB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**OR with Complement**

**X-form**

orc    RA,RS,RB
orc.   RA,RS,RB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of register RS are ANDed with the complement of the contents of register RB and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if Rc=1)

The contents of register RS are XORed with the complement of the contents of register RB and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if Rc=1)

### Extend Sign Byte

<table>
<thead>
<tr>
<th>Extendsb</th>
<th>RA,RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>extsb</td>
<td></td>
</tr>
<tr>
<td>extsb.</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s ← (RS)_{56}

RA_{56:63} ← (RS)_{56:63}

RA_{0:55} ← 56_{8}

(RS)_{56:63} are placed into RA_{56:63}. RA_{0:55} are filled with a copy of (RS)_{56}.

**Special Registers Altered:**

CR0 (if Rc=1)

### Extend Sign Halfword

<table>
<thead>
<tr>
<th>Extendsh</th>
<th>RA,RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>extsh</td>
<td></td>
</tr>
<tr>
<td>extsh.</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s ← (RS)_{48}

RA_{48:63} ← (RS)_{48:63}

RA_{0:47} ← 48_{8}

(RS)_{48:63} are placed into RA_{48:63}. RA_{0:47} are filled with a copy of (RS)_{48}.

**Special Registers Altered:**

CR0 (if Rc=1)

### Count Leading Zeros Word

<table>
<thead>
<tr>
<th>Cntlzw</th>
<th>RA,RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>cntlzw</td>
<td>RA,RS</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

Chapter 3. Fixed-Point Facility 85
cntlzwc. RA,RS (Rc=1)

\[
\begin{array}{cccccc}
31 & 30 & 29 & 28 & 27 & 26 & \text{Rc} \\
& & & & & & 31
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 32 & 33 & 34 & 35 & 36 & 37
\end{array}
\]

\[
\text{n} \leftarrow 32 \\
\text{do while } n < 64 \\
\quad \text{if } (\text{RS})_n = 1 \text{ then leave} \\
\quad n \leftarrow n + 1 \\
\quad \text{RA} \leftarrow n - 32
\]

A count of the number of consecutive zero bits starting at bit 32 of register RS is placed into register RA. This number ranges from 0 to 32, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

**Special Registers Altered:**
CR0  
(if Rc=1)

**Programming Note**
For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to 0 in CR Field 0.

### Compare Bytes

#### X-form

cmpb RA,RS,RB

\[
\begin{array}{cccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 \\
& & & & & & 31
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]

\[
\text{do } n = 0 \text{ to } 7 \\
\quad \text{if } RS_{8\times n:8\times n+7} = (RB)_{8\times n:8\times n+7} \text{ then} \\
\quad \quad \text{RA}_{8\times n:8\times n+7} \leftarrow 1 \\
\quad \text{else} \\
\quad \quad \text{RA}_{8\times n:8\times n+7} \leftarrow 0
\]

Each byte of the contents of register RS is compared to each corresponding byte of the contents in register RB.

If they are equal, the corresponding byte in RA is set to 0xFF. Otherwise the corresponding byte in RA is set to 0x00.

**Special Registers Altered:**
None
### Population Count Bytes

**X-form**

<table>
<thead>
<tr>
<th>popcntb</th>
<th>RA, RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
</tr>
</tbody>
</table>

```
\[
\begin{array}{llllll}
  & 31 & 8 & 11 & 16 & 21 & 122 \\
\end{array}
\]
```

```
do i = 0 to 7
  n ← 0
  do j = 0 to 7
    if (RS)(i×8)+j = 1 then
      n ← n+1
      RA(i×8):(i×8)+7 ← n
  RA[i×8]:[i×8]+7 ← n
```

A count of the number of one bits in each byte of register RS is placed into the corresponding byte of register RA. This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**

None

### Population Count Words

**X-form**

<table>
<thead>
<tr>
<th>popcntw</th>
<th>RA, RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
</tr>
</tbody>
</table>

```
\[
\begin{array}{llllll}
  & 31 & 8 & 11 & 16 & 21 & 378 \\
\end{array}
\]
```

```
do i = 0 to 1
  n ← 0
  do j = 0 to 31
    if (RS)(i×32)+j = 1 then
      n ← n+1
      RA(i×32):(i×32)+31 ← n
  RA[i×32]:[i×32]+31 ← n
```

A count of the number of one bits in each word of register RS is placed into the corresponding word of register RA. This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**

None
Parity Doubleword

\texttt{prtyd \texttt{RA,RS}}

\hspace{1em} [Category: 64-bit]

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>186</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

\textit{s} \leftarrow 0
\textbf{do} \ i = 0 \ \textbf{to} \ 7
\quad \textit{s} \leftarrow \textit{s} \oplus (\texttt{RS})_{16^7+7}
\texttt{RA} \leftarrow 63\|\textit{s}

The least significant bit in each byte of the contents of register \texttt{RS} is examined. If there is an odd number of one bits the value 1 is placed into register \texttt{RA}; otherwise the value 0 is placed into register \texttt{RA}.

\textbf{Special Registers Altered:}

None

---

Parity Word

\texttt{prtyw \texttt{RA,RS}}

\hspace{1em} | 31 | 6 | 11 | 16 | 21 | 154 | / |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textit{s} \leftarrow 0
\textbf{do} \ i = 0 \ \textbf{to} \ 3
\quad \textit{s} \leftarrow \textit{s} \oplus (\texttt{RS})_{16^7+7}
\textbf{do} \ i = 4 \ \textbf{to} \ 7
\quad \textit{t} \leftarrow \textit{t} \oplus (\texttt{RS})_{16^7+7}
\texttt{RA}_{0:31} \leftarrow 32\|\textit{s}
\texttt{RA}_{32:63} \leftarrow 32\|\textit{t}

The least significant bit in each byte of (\texttt{RS})_{0:31} is examined. If there is an odd number of one bits the value 1 is placed into \texttt{RA}_{0:31}; otherwise the value 0 is placed into \texttt{RA}_{0:31}. The least significant bit in each byte of (\texttt{RS})_{32:63} is examined. If there is an odd number of one bits the value 1 is placed into \texttt{RA}_{32:63}; otherwise the value 0 is placed into \texttt{RA}_{32:63}.

\textbf{Special Registers Altered:}

None

---

\textit{Programming Note}

The \textit{Parity} instructions are designed to be used in conjunction with the \textit{Population Count} instruction to compute the parity of words or a doubleword. The parity of the upper and lower words in (\texttt{RS}) can be computed as follows.

\texttt{popcntb RA, RS}
\texttt{prtyw RA, RA}

The parity of (\texttt{RS}) can be computed as follows.

\texttt{popcntb RA, RS}
\texttt{prtyd RA, RA}
### 3.3.13.1 64-bit Fixed-Point Logical Instructions [Category: 64-Bit]

#### Extend Sign Word X-form

<table>
<thead>
<tr>
<th>extsw</th>
<th>RA,RS</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>extsw.</td>
<td>RA,RS</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>986</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
s \leftarrow (RS)_{32}
\]

\[
RA_{32:63} \leftarrow (RS)_{32:63}
\]

\[
RA_{0:31} \leftarrow 32_s
\]

(RS)_{32:63} are placed into RA_{32:63}. RA_{0:31} are filled with a copy of (RS)_{32}.

**Special Registers Altered:**

CR0 (if Rc=1)

---

#### Count Leading Zeros Doubleword X-form

<table>
<thead>
<tr>
<th>cntizd</th>
<th>RA,RS</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cntizd.</td>
<td>RA,RS</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>58</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
n \leftarrow 0
\]

\[
do\ while\ n < 64
\]

\[
\text{if } (RS)_i = 1\ \text{then}
\]

\[
n \leftarrow n + 1
\]

\[
RA \leftarrow n
\]

A count of the number of consecutive zero bits starting at bit 0 of register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**

None

---

#### Population Count Doubleword X-form

<table>
<thead>
<tr>
<th>popcntd</th>
<th>RA,RS</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>506</th>
<th>/</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

\[
n \leftarrow 0
\]

\[
do\ i = 0\ \text{to} 63
\]

\[
\text{if } (RS)_i = 1\ \text{then}
\]

\[
n \leftarrow n + 1
\]

\[
RA \leftarrow n
\]

A count of the number of one bits in register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**

None
Bit Permute Doubleword  X-form

bpermd RA,RS,RB  
[Category: Embedded.Phased-in, Server]

|   | 31 | RS | RA | RB | 252 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

For i = 0 to 7
  index ← (RS)8*i:8*i+7
  If index < 64
    then perm_i ← (RB)_{index}
    else perm_i ← 0
  RA ← 560 || perm0:7

Eight permuted bits are produced. For each permuted bit i where i ranges from 0 to 7 and for each byte i of RS, do the following.

  If byte i of RS is less than 64, permuted bit i is set to the bit of RB specified by byte i of RS; otherwise permuted bit i is set to 0.

The permuted bits are placed in the least-significant byte of RA, and the remaining bits are filled with 0s.

Special Registers Altered:
  None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 63 permits the permuted bits to be selected from a 128-bit quantity, using a single index register. For example, assume that the 128-bit quantity Q, from which the permuted bits are to be selected, is in registers r2 (high-order 64 bits of Q) and r3 (low-order 64 bits of Q), that the index values are in register r1, with each byte of r1 containing a value in the range 0:127, and that each byte of register r4 contains the value 64. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of r6.

bpermd r6,r1,r2  # select from high-order half of Q
xor r0,r1,r4     # adjust index values
bpermd r5,r0,r3  # select from low-order half of Q
or r6,r6,r5      # merge the two selections
3.3.14 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Facility performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted rotate64 or ROTL64, the value rotated is the given 64-bit value. The rotate64 operation is used to rotate a given 64-bit quantity.

For the second type, denoted rotate32 or ROTL32, the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The rotate32 operation is used to rotate a given 32-bit quantity.

The Rotate and Shift instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, mstart, through and including a stop bit, mstop, and 0-bits elsewhere. The values of mstart and mstop range from 0 to 63. If mstart > mstop, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
    mask_mstart:mstop = ones
    mask_all other bits = zeros
else
    mask_mstart:63 = ones
    mask_0:mstop = ones
    mask_all other bits = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the rotate32 operation, the mask start and stop positions are always in the low-order 32 bits of the mask.

The use of the mask is described in following sections.

The Rotate and Shift instructions with Rc=1 set the first three bits of CR field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 65. Rotate and Shift instructions do not change the OV and SO bits. Rotate and Shift instructions, except algebraic right shifts, do not change the CA bit.

Extended mnemonics for rotates and shifts

The Rotate and Shift instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and performing simple rotates and shifts. Some of these are shown as examples with the Rotate instructions. See Appendix E, “Assembler Extended Mnemonics” on page 709 for additional extended mnemonics.

3.3.14.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or

- ANDed with a mask before being placed into the target register.

The Rotate Left instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64–n, where n is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32–n, where n is the number of bits by which to rotate right.

The contents of register RS are rotated32 left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)
Extended Mnemonics:

Examples of extended mnemonics for Rotate Left Word Immediate then AND with Mask:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>extlwi Rx,Ry,n,b</td>
<td>rlwinm Rx,Ry,b,0,n-1</td>
</tr>
<tr>
<td>snwi Rx,Ry,n</td>
<td>rlwinm Rx,Ry,32-n,n,31</td>
</tr>
<tr>
<td>clrwi Rx,Ry,n</td>
<td>rlwinm Rx,Ry,0,0,31-n</td>
</tr>
</tbody>
</table>

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

**rlwinm** can be used to extract an n-bit field that starts at bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b, MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at bit position b in RSL, left-justified into the low-order 32 bits of RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b, MB=0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by n bits, by setting SH=n bits, by setting SH=n (32-n), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by n bits, by setting SH=32-n, MB=0, and ME=31. It can be used to clear the high-order b bits of the low-order 32 bits of the contents of a register and then shift the result left by n bits, by setting SH=n, MB=b, and ME=31-n. It can be used to clear the low-order n bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31-n. For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for all of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

**Rotate Left Word then AND with Mask**

M-form

<table>
<thead>
<tr>
<th></th>
<th>RA,RS,MB,ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlwinm</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>rlwinm.</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>23</td>
<td>26</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>b</td>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>MB</td>
</tr>
</tbody>
</table>

\[ n \leftarrow (RB)_{59:63} \]
\[ r \leftarrow ROTL_{32}(RS)_{32:63}, n \]
\[ m \leftarrow MASK(MB+32, ME+32) \]
\[ RA \leftarrow r \& m \]

The contents of register RS are rotated_32 left the number of bits specified by (RB)_{59:63}. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDeD with the generated mask and the result is placed into register RA.

Special Registers Altered:

<table>
<thead>
<tr>
<th></th>
<th>CR0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(if Rc=1)</td>
</tr>
</tbody>
</table>

Extended Mnemonics:

Example of extended mnemonics for Rotate Left Word then AND with Mask:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotlw Rx,Ry,Rz</td>
<td>rlwinm Rx,Ry,0,31</td>
</tr>
</tbody>
</table>

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

**rlwinm** can be used to extract an n-bit field that starts at variable bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB_{59:63}=b, MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at variable bit position b in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB_{59:63}=b, MB=0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by variable n bits, by setting RB_{59:63}=n (32-n), MB=0, and ME=31. For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for some of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.
Rotate Left Word Immediate then Mask Insert

\[ \text{rlwimi } RA,RS,SH,MB,ME \quad (Rc=0) \]

\[ \text{rlwimi. } RA,RS,SH,MB,ME \quad (Rc=1) \]

\[
\begin{array}{ccccccc}
20 & RS & RA & SH & MB & ME & Rc \\
0 & b & 11 & 16 & 21 & 26 & 31 \\
\end{array}
\]

\[
\begin{align*}
 n & \leftarrow SH \\
r & \leftarrow \text{ROTL}_{32}(\text{RS}_{32:63}, n) \\
m & \leftarrow \text{MASK}(MB+32, ME+32) \\
RA & \leftarrow r \& m \mid (RA) \& \neg m
\end{align*}
\]

The contents of register RS are rotated left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 \quad \text{(if } Rc=1) \]

Extended Mnemonics:

Example of extended mnemonics for Rotate Left Word Immediate then Mask Insert:

Extended: \hspace{1cm} Equivalent to:

\[
\begin{align*}
\text{inSlwi} & \quad Rx,Ry,n,b \quad \text{rlwimi } Rx,Ry,32-b,b,b+n-1
\end{align*}
\]

Programming Note

Let RAL represent the low-order 32 bits of register RA, with the bits numbered from 0 through 31.

\text{rlwimi} can be used to insert an n-bit field that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting \( SH=32-b, MB=b, \) and \( ME=(b+n)-1 \). It can be used to insert an n-bit field that is right-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting \( SH=32-(b+n), MB=b, \) and \( ME=(b+n)-1 \).

Extended mnemonics are provided for both of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.
3.3.14.1.1 64-bit Fixed-Point Rotate Instructions [Category: 64-Bit]

### Rotate Left Doubleword Immediate then Clear Left

<table>
<thead>
<tr>
<th>MD-form</th>
<th>RA,RS,SH,MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldicl</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>rldicl.</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RS</th>
<th>RA</th>
<th>sh</th>
<th>mb</th>
<th>0</th>
<th>sh</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>27</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
n \leftarrow sh_0 || sh_{0:4} \\
r \leftarrow \text{ROTL}_{64}((RS), n) \\
b \leftarrow mb_5 || mb_{0:4} \\
m \leftarrow \text{MASK}(b, 63) \\
RA \leftarrow r \& m
\]

The contents of register RS are rotated \(64\) left \(SH\) bits. A mask is generated having 1-bits from bit \(MB\) through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**
- CR0 (if Rc=1)

**Extended Mnemonics:**

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Left*:

- **Extended:**
  - extrdi Rx,Ry,n,b
  - srdi Rx,Ry,n
  - clrldi Rx,Ry,n

- **Equivalent to:**
  - rldicl Rx,Ry,b+n,64−n
  - rldicl Rx,Ry,64−n

**Programming Note**

*rldicl* can be used to extract an \(n\)-bit field that starts at bit position \(b\) in register RS, right-justified into register RA (clearing the remaining \(64−n\) bits of RA), by setting \(SH=b+n\) and \(MB=64−n\). It can be used to rotate the contents of a register left (right) by \(n\) bits, by setting \(SH=n\) (64−n) and \(MB=0\). It can be used to shift the contents of a register right by \(n\) bits, by setting \(SH=64−n\) and \(MB=n\). It can be used to clear the high-order \(n\) bits of a register, by setting \(SH=0\) and \(MB=n\).

Extended mnemonics are provided for all of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

### Rotate Left Doubleword Immediate then Clear Right

<table>
<thead>
<tr>
<th>MD-form</th>
<th>RA,RS,SH,ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldicr</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>rldicr.</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RS</th>
<th>RA</th>
<th>sh</th>
<th>me</th>
<th>1</th>
<th>sh</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>27</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
n \leftarrow sh_5 || sh_{0:4} \\
r \leftarrow \text{ROTL}_{64}((RS), n) \\
e \leftarrow me_5 || me_{0:4} \\
m \leftarrow \text{MASK}(0, e) \\
RA \leftarrow r \& m
\]

The contents of register RS are rotated \(64\) left \(SH\) bits. A mask is generated having 1-bits from bit 0 through bit \(ME\) and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**
- CR0 (if Rc=1)

**Extended Mnemonics:**

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Right*:

- **Extended:**
  - extldi Rx,Ry,n,b
  - sldi Rx,Ry,n
  - clrrdi Rx,Ry,n

- **Equivalent to:**
  - rldicr Rx,Ry,b,n−1
  - rldicr Rx,Ry,63−n

**Programming Note**

*rldicr* can be used to extract an \(n\)-bit field that starts at bit position \(b\) in register RS, left-justified into register RA (clearing the remaining \(64−n\) bits of RA), by setting \(SH=b\) and \(ME=n−1\). It can be used to rotate the contents of a register left (right) by \(n\) bits, by setting \(SH=n\) (64−n) and \(ME=63\). It can be used to shift the contents of a register left by \(n\) bits, by setting \(SH=n\) and \(ME=63−n\). It can be used to clear the low-order \(n\) bits of a register, by setting \(SH=0\) and \(ME=63−n\).

Extended mnemonics are provided for all of these uses (some devolve to *rldicl*); see Appendix E, “Assembler Extended Mnemonics” on page 709.
**Rotate Left Doubleword Immediate then Clear**

*MD-form*

<table>
<thead>
<tr>
<th>rldic</th>
<th>RA,RS,SH,MB (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldic.</td>
<td>RA,RS,SH,MB (Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30</th>
<th>RS</th>
<th>RA</th>
<th>sh</th>
<th>mb</th>
<th>2</th>
<th>sh</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>27</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r & m

The contents of register RS are rotated left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**

- CR0 (if Rc=1)

**Extended Mnemonics:**

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Clear*:

**Extended:**

<table>
<thead>
<tr>
<th>cllsldi</th>
<th>Rx,Ry,b,n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to:</td>
<td>rldic Rx,Ry,n,b−n</td>
</tr>
</tbody>
</table>

**Programming Note**

*rldic* can be used to clear the high-order b bits of the contents of a register and then shift the result left by n bits, by setting SH=n and MB=b−n. It can be used to clear the high-order n bits of a register, by setting SH=0 and MB=n.

Extended mnemonics are provided for both of these uses (the second devolves to *rldic*); see Appendix E, “Assembler Extended Mnemonics” on page 709.

**Rotate Left Doubleword then Clear Left**

*MDS-form*

<table>
<thead>
<tr>
<th>rldc</th>
<th>RA,RS,RB,MB (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldcl</td>
<td>RA,RS,RB,MB (Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>mb</th>
<th>8</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>27</td>
<td>31</td>
</tr>
</tbody>
</table>

n ← (RB)58:63
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m

The contents of register RS are rotated left the number of bits specified by (RB)58:63. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**

- CR0 (if Rc=1)

**Extended Mnemonics:**

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left*:

**Extended:**

<table>
<thead>
<tr>
<th>cllsldi</th>
<th>Rx,Ry,b,n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to:</td>
<td>rldic Rx,Ry,n,b−n</td>
</tr>
</tbody>
</table>

**Programming Note**

*rldc* can be used to extract an n-bit field that starts at variable bit position b in register RS, right-justified into register RA (clearing the remaining 64−n bits of RA), by setting RB58:63=b+n and MB=64−n. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB58:63=n (64−n) and MB=0.

Extended mnemonics are provided for some of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.
### Rotate Left Doubleword then Clear Right

**MDS-form**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>8</td>
<td>16</td>
<td>21</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>me</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rc</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
n & \leftarrow (RB)_{58:63} \\
r & \leftarrow \text{ROTL}_{64}(RS, n) \\
e & \leftarrow \text{me} \_ \text{e} \_ \text{m} \\
m & \leftarrow \text{MASK}(0, e) \\
RA & \leftarrow r \_ \& \_ m
\end{align*}
\]

The contents of register RS are rotated left the number of bits specified by \((RB)_{58:63}\). A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**

- CR0

**Programming Note**

\textit{rldcr} can be used to extract an n-bit field that starts at variable bit position b in register RS, left-justified into register RA (clearing the remaining 64-n bits of RA), by setting RB\(_{58:63}\)=b and ME=n-1. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB\(_{58:63}\)=n (64-n) and ME=63.

Extended mnemonics are provided for some of these uses (some devolve to \textit{rldcl}); see Appendix E, “Assembler Extended Mnemonics” on page 709.

### Rotate Left Doubleword Immediate then Mask Insert

**MD-form**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>RA</td>
<td>sh</td>
<td>mb</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sh</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rc</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
n & \leftarrow \text{sh} \_ \text{e} \_ \text{m} \\
r & \leftarrow \text{ROTL}_{64}(RS, n) \\
b & \leftarrow \text{mb} \_ \text{e} \_ \text{m} \\
m & \leftarrow \text{MASK}(b, \neg n) \\
RA & \leftarrow r \_ \& \_ m \_ (RA) \_ \& \_ \neg m
\end{align*}
\]

The contents of register RS are rotated left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

**Special Registers Altered:**

- CR0

**Programming Note**

\textit{rldimi} can be used to insert an n-bit field that is right-justified in register RS, into register RA starting at bit position b, by setting SH=64-(b+n) and MB=b.

An extended mnemonic is provided for this use; see Appendix E, “Assembler Extended Mnemonics” on page 709.
3.3.14.2 Fixed-Point Shift Instructions

The instructions in this section perform left and right shifts.

**Extended mnemonics for shifts**

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain *Rotate* instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand. Some of these are shown as examples with the *Rotate* instructions. See Appendix E, “Assembler Extended Mnemonics” on page 709 for additional extended mnemonics.

---

**Programming Note**

Any Shift Right Algebraic instruction, followed by *addze*, can be used to divide quickly by $2^n$. The setting of the CA bit by the Shift Right Algebraic instructions is independent of mode.

**Programming Note**

Multiple-precision shifts can be programmed as shown in Section F.1, “Multiple-Precision Shifts” on page 723.

---

### Shift Left Word

**X-form**

\[
\text{slw} \quad \text{RA,RS,RB} \quad \text{(Rc=0)} \\
\text{slw.} \quad \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>(RB)59:63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>$\text{ROTL}<em>{32}(\text{RS})</em>{32:63}, n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (RB)58 = 0 then</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>$\text{MASK}(32, 63-n)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else m</td>
<td>640</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>r &amp; m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by (RB)58:63. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $\text{RA}_{32:63}$: $\text{RA}_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

**Special Registers Altered:**

CR0 (if Rc=1)

### Shift Right Word

**X-form**

\[
\text{srw} \quad \text{RA,RS,RB} \quad \text{(Rc=0)} \\
\text{srw.} \quad \text{RA,RS,RB} \quad \text{(Rc=1)}
\]

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>536</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>(RB)59:63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>$\text{ROTL}<em>{32}(\text{RS})</em>{32:63}, 64-n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (RB)58 = 0 then</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>$\text{MASK}(n+32, 63)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else m</td>
<td>640</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>r &amp; m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by (RB)58:63. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $\text{RA}_{32:63}$: $\text{RA}_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

**Special Registers Altered:**

CR0 (if Rc=1)
**Shift Right Algebraic Word Immediate X-form**

srawi  RA,RS,SH  \( \text{(Rc}=0) \)

srawi.  RA,RS,SH  \( \text{(Rc}=1) \)

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>SH</th>
<th>824</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

\( n \leftarrow \text{SH} \)

\( r \leftarrow \text{ROTL}_{32}((RS)_{32:63}, 64-n) \)

\( m \leftarrow \text{MASK}(n+32, 63) \)

\( s \leftarrow (RS)_{32} \)

\( RA \leftarrow r \& m \mid (64s)\&\neg m \)

\( CA \leftarrow s \& ((r\&\neg m)_{32:63}\neq 0) \)

The contents of the low-order 32 bits of register RS are shifted right \( SH \) bits. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into RA\(_{32:63}\). Bit 32 of RS is replicated to fill RA\(_{0:31}\). CA is set to 0 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive EXTS((RS)\(_{32:63}\)), and CA to be set to 0.

**Special Registers Altered:**

- CA
- CR0

**Shift Right Algebraic Word X-form**

sraw  RA,RS,RB  \( \text{(Rc}=0) \)

sraw.  RA,RS,RB  \( \text{(Rc}=1) \)

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>792</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

\( n \leftarrow (RB)_{59:63} \)

\( r \leftarrow \text{ROTL}_{32}((RS)_{32:63}, 64-n) \)

if \( (RB)_{58} = 0 \) then

\( m \leftarrow \text{MASK}(n+32, 63) \)

else \( m \leftarrow 640 \)

\( s \leftarrow (RS)_{32} \)

\( RA \leftarrow r \& m \mid (64s)\&\neg m \)

\( CA \leftarrow s \& ((r\&\neg m)_{32:63}\neq 0) \)

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by \((RB)_{58:63}\). Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into RA\(_{32:63}\). Bit 32 of RS is replicated to fill RA\(_{0:31}\). CA is set to 0 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive EXTS((RS)\(_{32:63}\)), and CA to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive the sign bit of (RS)\(_{32:63}\).

**Special Registers Altered:**

- CA
- CR0

(if Rc=1)
3.3.14.2.1 64-bit Fixed-Point Shift Instructions
[Category: 64-Bit]

**Shift Left Doubleword**

<table>
<thead>
<tr>
<th>X-form</th>
<th>sld</th>
<th>RA,RS,RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sld.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 27 & Rc \\
0 & 31 & 6 & 11 & 16 & 21 & 31 & 539 & Rc
\end{array}
\]

- \( n \leftarrow (RB)_{58:63} \)
- \( r \leftarrow \text{ROTL}_{64}(RS, n) \)
- if \((RB)_{57} = 0\) then
  - \( m \leftarrow \text{MASK}(0, 63-n) \)
- else \( m \leftarrow 640 \)
- \( RA \leftarrow r \& m \)

The contents of register RS are shifted left the number of bits specified by \((RB)_{57:63}\). Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

**Special Registers Altered:**
- CR0 (if Rc=1)

**Shift Right Doubleword**

<table>
<thead>
<tr>
<th>X-form</th>
<th>srd</th>
<th>RA,RS,RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>srd.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 27 & Rc \\
0 & 31 & 6 & 11 & 16 & 21 & 31 & 539 & Rc
\end{array}
\]

- \( n \leftarrow (RB)_{58:63} \)
- \( r \leftarrow \text{ROTL}_{64}(RS, 64-n) \)
- if \((RB)_{57} = 0\) then
  - \( m \leftarrow \text{MASK}(n, 63) \)
- else \( m \leftarrow 640 \)
- \( RA \leftarrow r \& m \)

The contents of register RS are shifted right the number of bits specified by \((RB)_{57:63}\). Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

**Special Registers Altered:**
- CR0 (if Rc=1)
**Shift Right Algebraic Doubleword**  
**Immediate XS-form**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>413</th>
<th>sh</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
n & \leftarrow \text{sh}_5 \ || \ \text{sh}_0:4 \\
r & \leftarrow \text{ROTL}_{64}(\text{RS}, \ 64-n) \\
m & \leftarrow \text{MASK}(n, \ 63) \\
s & \leftarrow (\text{RS})_0 \\
\text{RA} & \leftarrow r \& m \ | \ (64s) \& \neg m \\
\text{CA} & \leftarrow s \ & ((r \& \neg m) \neq 0)
\end{align*}
\]

The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0.

**Special Registers Altered:**
- \(\text{CA}\)
- \(\text{CR0}\) (if Rc=1)

**Shift Right Algebraic Doubleword**  
**X-form**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
n & \leftarrow (\text{RB})_58:63 \\
r & \leftarrow \text{ROTL}_{64}(\text{RS}, \ 64-n) \\
\text{if} \ (\text{RB})_{57} = 0 \ & \text{then} \\
\quad m & \leftarrow \text{MASK}(n, \ 63) \\
\text{else} \ m & \leftarrow 640 \\
\text{RA} & \leftarrow r \& m \ | \ (64s) \& \neg m \\
\text{CA} & \leftarrow s \ & ((r \& \neg m) \neq 0)
\end{align*}
\]

The contents of register RS are shifted right the number of bits specified by \((\text{RB})_{57:63}\). Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA to receive the sign bit of (RS).

**Special Registers Altered:**
- \(\text{CA}\)
- \(\text{CR0}\) (if Rc=1)
3.3.15 Binary Coded Decimal (BCD) Assist Instructions [Category: Embedded.Phased-in, Server]

The Binary Coded Decimal Assist instructions operate on Binary Coded Decimal operands (cbcdtd and cdbcd) and Decimal Floating-Point operands (cdt-bcd) See Chapter 6. for additional information.

**Convert Declets To Binary Coded Decimal X-form**

```
cdbcd RA, RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>--</th>
<th>282</th>
<th>--</th>
</tr>
</thead>
</table>
```

`do i = 0 to 1
n ← i x 32
RA_{n+0:n+7} ← 0
RA_{n+8:n+19} ← DPD_TO_BCD( (RS)_{n+12:n+21} )
RA_{n+20:n+31} ← DPD_TO_BCD( (RS)_{n+22:n+31} )`

The low-order 20 bits of each word of register RS contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the corresponding word in RA. The high-order 8 bits in each word of RA are set to 0.

**Special Registers Altered:**
None

**Convert Binary Coded Decimal To Declets X-form**

```
ccbddd RA, RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>--</th>
<th>314</th>
<th>--</th>
</tr>
</thead>
</table>
```

`do i = 0 to 1
n ← i x 32
RA_{n+0:n+11} ← 0
RA_{n+12:n+21} ← BCD_TO_DPD( (RS)_{n+8:n+19} )
RA_{n+22:n+31} ← BCD_TO_DPD( (RS)_{n+20:n+31} )`

The low-order 24 bits of each word of register RS contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the corresponding word in RA. The high-order 12 bits in each word of RA are set to 0.

**Special Registers Altered:**
None

**Add and Generate Sixes XO-form**

```
addg6s RT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>--</th>
<th>74</th>
<th>--</th>
</tr>
</thead>
</table>
```

`do i = 0 to 15
dc_i ← carry_out( RA_{4xi:63} + RB_{4xi:63} )
c ← 4(dc_0) || 4(dc_1) || ... || 4(dc_{15})
RT ← (¬c) & 0x6666_6666_6666_6666`

The contents of register RA are added to the contents of register RB. Sixteen carry bits are produced, one for each carry out of decimal position n (bit position 4xn).

A doubleword is composed from the 16 carry bits, and placed into RT. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding carry bit is 0, and a zero (0b0000) in every position for which the corresponding carry bit is 1.

**Special Registers Altered:**
None


addg6s can be used to add or subtract two BCD operands. In these examples it is assumed that r0 contains 0x666...666. (BCD data formats are described in Section 6.3.)

Addition of the unsigned BCD operand in register RA to the unsigned BCD operand in register RB can be accomplished as follows.

\[
\begin{align*}
\text{add} & \quad r1, RA, r0 \\
\text{add} & \quad r2, r1, RB \\
\text{addg6s} & \quad RT, r1, RB \\
\text{subf} & \quad RT, RT, r2 \# RT = RA \times_{BCD} RB
\end{align*}
\]

Subtraction of the unsigned BCD operand in register RA from the unsigned BCD operand in register RB can be accomplished as follows. (In this example it is assumed that RB is not register 0.)

\[
\begin{align*}
\text{addi} & \quad r1, RB, 1 \\
\text{nor} & \quad r2, RA, RA \# \text{one's complement of RA} \\
\text{add} & \quad r3, r1, r2 \\
\text{addg6s} & \quad RT, r1, r2 \\
\text{subf} & \quad RT, RT, r3 \# RT = RB -_{BCD} RA
\end{align*}
\]

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).
3.3.16 Move To/From System Register Instructions

The *Move To Condition Register Fields* instruction has a preferred form; see Section 1.8.1, "Preferred Instruction Forms" on page 22. In the preferred form, the FXM field satisfies the following rule:

- Exactly one bit of the FXM field is set to 1.

**Extended mnemonics**

Extended mnemonics are provided for the *mtspr* and *mfspr* instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. An extended mnemonic is provided for the *mtcrf* instruction for compatibility with old software (written for a version of the architecture that precedes Version 2.00) that uses it to set the entire Condition Register. Some of these extended mnemonics are shown as examples with the relevant instructions. See Appendix E, "Assembler Extended Mnemonics" on page 709 for additional extended mnemonics.
Move To Special Purpose Register
XFX-form

mtspr SPR,RS

\[
\begin{array}{cccc}
\text{decimal} & \text{SPR}^1 spr_{5:9} & \text{SPR}_{0:4} & \text{Name} \\
\hline
1 & 00000 00001 & XER & \\
3 & 00000 00011 & DSCR^3 & \\
8 & 00000 01000 & LR & \\
9 & 00000 01001 & CTR & \\
13 & 00000 01101 & AMR^3 & \\
128 & 00100 00000 & TFHAR^4 & \\
129 & 00100 00001 & TFIAR^4 & \\
130 & 00100 00010 & TEXASR^4 & \\
131 & 00100 00011 & TEXASRU^4 & \\
256 & 01000 00000 & VRSAVE & \\
512 & 10000 00000 & SPEFSCR^2 & \\
769 & 11000 00001 & MMCRC1^3 & \\
771 & 11000 00011 & MMCRA^3 & \\
\hline
\end{array}
\]

1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. Category: SPE.
3. Category: Server; see Book III-S.
4. Category: Transactional Memory. See Chapter 5 of Book II.
5. Category: Stream.
6. Accesses to these registers are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”
7. Category: Server; see Book II.
8. Accesses to these registers are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”
9. Category: Server; see Book II.

If execution of this instruction is attempted specifying an SPR number that is not shown above, or an SPR number that is shown above but is in a category that is not supported by the implementation, one of the following occurs:

- If spr0 = 0, the illegal instruction error handler is invoked.
- If spr0 = 1, the system privileged instruction error handler is invoked.

If an attempt is made to execute \texttt{mtspr} specifying a TM SPR in other than Non-Transactional state, a TM Bad Thing type Program interrupt is generated.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

See above

Extended Mnemonics:

Examples of extended mnemonics for \texttt{Move To Special Purpose Register}:

\[
\begin{array}{ll}
\text{Extended} & \text{Equivalent to} \\
\texttt{mtxer} Rx & \texttt{mtspr} 1,Rx \\
\texttt{mtlr} Rx & \texttt{mtspr} 8,Rx \\
\end{array}
\]
Extended:               Equivalent to:
mtctr    Rx               mtspr   9,Rx
mtppr    Rx               mtspr  896,Rx
mtppr32  Rx               mtspr  898,Rx

--- Programming Note ---
The AMR is part of the “context” of the program  
(see Book III-S).  Therefore modification of the AMR  
requires “synchronization” by software.  For this rea-  
son, most operating systems provide a system  
library program that application programs can use  
to modify the AMR.

--- Compiler and Assembler Note ---
For the mtspr and mfspr instructions, the SPR  
number coded in Assembler language does not  
appear directly as a 10-bit binary number in the  
instruction. The number coded is split into two 5-bit  
halves that are reversed in the instruction, with the  
high-order 5 bits appearing in bits 16:20 of the  
instruction and the low-order 5 bits in bits 11:15.
Move From Special Purpose Register

**XFX-form**

```markdown
mfspr  RT,SPR
```

```markdown
n <- spr5:9 || spr0:4
switch (n)
  case(129): see Book III-S
  case(808, 809, 810, 811):
    default:
      if length(SPR[n]) = 64 then
        RT <- SPR[n]
      else
        RT <- 32
        SPRT[n]
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains 129, the instruction references the Transaction Failure Instruction Address Register (TFIAR) and the result is dependent on the privilege with which it is executed. See Book III-S. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a noop; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

![Register Name Table]

### Note
1. Note that the order of the two 5-bit halves of the SPR number is reversed.
2. Category: Embedded.
3. See Chapter 6 of Book II.
4. Category: SPE.
6. Category: Server; see Book III-S.
7. Category: Transactional Memory. See Chapter 5 of Book II.
9. Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”.
10. Category: Server; see Book II.

If execution of this instruction is attempted specifying an SPR number that is not shown above, or an SPR number that is shown above but is in a category that is not supported by the implementation, one of the following occurs.
If spr0 = 0, the illegal instruction error handler is invoked.
If spr0 = 1, the system privileged instruction error handler is invoked.

A complete description of this instruction can be found in Book III.

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Move From Special Purpose Register:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfxer Rx</td>
<td>mfspr Rx,1</td>
</tr>
<tr>
<td>mflr Rx</td>
<td>mfspr Rx,8</td>
</tr>
<tr>
<td>mfctr Rx</td>
<td>mfspr Rx,9</td>
</tr>
</tbody>
</table>

**Note**
See the Notes that appear with mtspcr.

*Move To Condition Register Fields*

**XFX-form**

```plaintext
mtcrf FXM,RS
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>0</th>
<th>FXM</th>
<th>/</th>
<th>144</th>
<th>/</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>2</td>
<td></td>
<td>20</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

mask = 4(FXM0) || 4(FXM1) || ... 4(FXM7)
CR = ((RS)32:63 & mask) | (CR & ¬mask)

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FXMi=1 then CR field i (CR bits 4×i+32:4×i+35) is set to the contents of the corresponding field of the low-order 32 bits of RS.

**Special Registers Altered:**
CR fields selected by mask

**Extended Mnemonics:**
Example of extended mnemonics for *Move To Condition Register Fields*:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtcrr Rx</td>
<td>mtcrr 0xFF,Rx</td>
</tr>
</tbody>
</table>

**Programming Note**
In the preferred form of this instruction (*mtocrf*), only one Condition Register field is updated.
### Move From Condition Register  XFX-form

**mfcr RT**

| 0 | 31 | 6 | 11 | 12 | 21 | 19 | 31 |

RT ← \(32 \_0 | CR\)

The contents of the Condition Register are placed into RT\(_{32:63}\). RT\(_{0:31}\) are set to 0.

**Special Registers Altered:**

None

### Move To Split Little Endian  X-form

**mtsle L**

[Category: Server]

| 31 | /// | L | /// | /// | 147 |

SLE ← L

The value specified in the L field is placed into SLE.

**L=0**

SLE is set to 0. The Endian mode used for both data and instruction storage accesses is represented by the value specified by LE.

**L=1**

SLE is set to 1. The Endian mode used for instruction storage accesses is represented by the value of LE. The Endian mode used for data storage accesses is represented by the value of ¬LE.

The effects of executing **mtsle** are immediately effective for subsequent storage accesses.

**Special Registers Altered:**

MSR (see Book III-S)
Move From VSR Doubleword XX1-form
[Category: Vector-Scalar]

mfvsrd RA,XS

<table>
<thead>
<tr>
<th>S</th>
<th>RA</th>
<th>///</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XS ← SX || S
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
GPR[RA] ← VSR[XS].doubleword[0]

Let XS be the value SX concatenated with S.

The contents of doubleword element 0 of VSR[XS] are placed into GPR[RA].

For SX=0, mfvsrd is treated as a Floating-Point instruction in terms of resource availability.

For SX=1, mfvsrd is treated as a Vector instruction in terms of resource availability.

Extended Mnemonics Equivalent To
mfprd RA,FRS mfvsrd RA,FRS
mfvrds RA,FRS+32

Special Registers Altered
None

Data Layout for mfvsrd
src = VSR[XS]

tgt = GPR[RA]

Move From VSR Word and Zero XX1-form
[Category: Vector-Scalar]

mfvsrwz RA,XS

<table>
<thead>
<tr>
<th>S</th>
<th>RA</th>
<th>///</th>
<th>115</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XS ← SX || S
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
GPR[RA] ← ExtendZero(VSR[XS].word[1])

XS ← SX || S
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
GPR[RA] ← ExtendZero(VSR[XS].word[1])

Let XS be the value SX concatenated with S.

The contents of word element 1 of VSR[XS] are placed into bits 32:63 of GPR[RA]. The contents of bits 0:31 of GPR[RA] are set to 0.

For SX=0, mfvsrwz is treated as a Floating-Point instruction in terms of resource availability.

For SX=1, mfvsrwz is treated as a Vector instruction in terms of resource availability.

Extended Mnemonics Equivalent To
mfprwz RA,FRS mfvsrwz RA,FRS
mfvrwz RA,FRS+32

Special Registers Altered
None

Data Layout for mfvsrwz
src = VSR[XS]

tgt = GPR[RA]
**Move To VSR Doubleword XX1-form**

[Category: Vector-Scalar]

```
mtvsrd XT,RA
```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>16</th>
<th>21</th>
<th>179</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

`XT ← TX || T`

- `if TX=0 & MSR.FP=0 then FP_Unavailable()`
- `if TX=1 & MSR.VEC=0 then Vector_Unavailable()`

- `VSR[XT].doubleword[0] ← GPR[RA]`
- `VSR[XT].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU`

Let `XT` be the value `TX` concatenated with `T`.

The contents of `GPR[RA]` are placed into doubleword element 0 of `VSR[XT]`.

The contents of doubleword element 1 of `VSR[XT]` are undefined.

For `TX=0`, `mtvsrd` is treated as a Floating-Point instruction in terms of resource availability.

For `TX=1`, `mtvsrd` is treated as a Vector instruction in terms of resource availability.

**Extended Mnemonics**

```
mtfprd FRT,RA
mtvrd VRT,RA
```

**Equivalent To**

```
mtvsrd FRT,RA
mtvsrd VRT+32,RA
```

**Special Registers Altered**

None

**Data Layout for mtvsrd**

- `src = GPR[RA]`
- `tgt = VSR[XT]`

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>64</th>
<th></th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Move To VSR Word Algebraic XX1-form**

[Category: Vector-Scalar]

```
mtvsrwa XT,RA
```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>16</th>
<th>21</th>
<th>211</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

`XT ← TX || T`

- `if TX=0 & MSR.FP=0 then FP_Unavailable()`
- `if TX=1 & MSR.VEC=0 then Vector_Unavailable()`

- `VSR[XT].doubleword[0] ← ExtendSign(GPR[RA].bit[32:63])`
- `VSR[XT].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU`

Let `XT` be the value `TX` concatenated with `T`.

The two's-complement integer in bits 32:63 of `GPR[RA]` is sign-extended to 64 bits and placed into doubleword element 0 of `VSR[XT]`.

The contents of doubleword element 1 of `VSR[XT]` are undefined.

For `TX=0`, `mtvsrwa` is treated as a Floating-Point instruction in terms of resource availability.

For `TX=1`, `mtvsrwa` is treated as a Vector instruction in terms of resource availability.

**Extended Mnemonics**

```
mtfprwa FRT,RA
mtvwrwa VRT,RA
```

**Equivalent To**

```
mtvsrwa FRT,RA
mtvsrwa VRT+32,RA
```

**Special Registers Altered**

None

**Data Layout for mtvsrwa**

- `src = GPR[RA]`
- `tgt = VSR[XT]`

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>32</th>
<th></th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

110  Power ISA™ - Book I
Move To VSR Word and Zero XX1-form
[Category: Vector-Scalar]

mtvsrwz XT, RA

Let XT be the value TX concatenated with T.

The contents of bits 32:63 of GPR[RA] are placed into word element 1 of VSR[XT]. The contents of word element 0 of VSR[XT] are set to 0.

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, mtvsrwz is treated as a Floating-Point instruction in terms of resource availability.

For TX=1, mtvsrwz is treated as a Vector instruction in terms of resource availability.

Extended Mnemonics Equivalent To
mtfprwz FRT, RA mtvsrwz FRT, RA
mtvrrwz VRT, RA mtvsrwz VRT+32, RA

Special Registers Altered
None

Data Layout for mtvsrwz

src = GPR[RA]

undefined

tgt = VSR[XT]

undefined

0 32 64 127
3.3.16.1 Move To/From One Condition Register Field Instructions

**Move To One Condition Register Field**  
*XFX-form*

**mtocrf**  
FXM,RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>1</th>
<th>FXM</th>
<th>/</th>
<th>144</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>12</td>
<td>/</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

count ← 0  
do i = 0 to 7  
    if FXM_i = 1 then  
        n ← i  
        count ← count + 1  
    if count = 1 then  
        CR_{4n+32:4n+35} ← (RS)_{4n+32:4n+35}  
    else CR ← undefined

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field (0 ≤ n ≤ 7). The contents of bits 4×n+32:4×n+35 of register RS are placed into CR field n (CR bits 4×n+32:4×n+35). Otherwise, the contents of the Condition Register are undefined.

**Special Registers Altered:**  
CR field selected by FXM

**Programming Note**

These forms of the `mtocrf` and `mfocrf` instructions are intended to replace the old forms of the instructions (the forms shown in page 107), which will eventually be phased out of the architecture. The new forms are backward compatible with most processors that comply with versions of the architecture that precede Version 2.00. On those processors, the new forms are treated as the old forms.

However, on some processors that comply with versions of the architecture that precede Version 2.00 the new forms may be treated as follows:

- **mtocrf:** may cause the system illegal instruction error handler to be invoked
- **mfocrf:** may place an undefined value into register RT

**Move From One Condition Register Field**  
*XFX-form*

**mfocrf**  
RT,FXM

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>1</th>
<th>FXM</th>
<th>/</th>
<th>19</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>12</td>
<td>/</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

RT ← undefined  
count ← 0  
do i = 0 to 7  
    if FXM_i = 1 then  
        n ← i  
        count ← count + 1  
    if count = 1 then  
        RT_{4n+32:4n+35} ← CR_{4n+32:4n+35}

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field (0 ≤ n ≤ 7). The contents of CR field n (CR bits 4×n+32:4×n+35) are placed into bits 4×n+32:4×n+35 of register RT and the contents of the remaining bits of register RT are undefined. Otherwise, the contents of register RT are undefined.

**Special Registers Altered:**  
None
### Move To/From System Registers [Category: Embedded]

#### Move to Condition Register from XER

**X-form**

```plaintext
mcrxr  BF
```

<table>
<thead>
<tr>
<th>31</th>
<th>BF</th>
<th>///</th>
<th>///</th>
<th>512</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

CR4×BF+32:4×BF+35 ← XER32:35
XER32:35 ← 0b0000

The contents of XER32:35 are copied to Condition Register field BF. XER32:35 are set to zero.

**Special Registers Altered:**
- CR field BF  XER32:35

#### Move To Device Control Register

**User-mode Indexed**

```plaintext
mtdcrux  RS,RA
```

[Category: Embedded.Device Control]

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>///</th>
<th>419</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT.

See “Move To Device Control Register Indexed X-form” on page 1055 in Book III for more information on this instruction.

**Special Registers Altered:**
- Implementation-dependent

#### Move From Device Control Register

**User-mode Indexed**

```plaintext
mfdcrux  RT,RA
```

[Category: Embedded.Device Control]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>RA</th>
<th>///</th>
<th>291</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into RT. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT.

See “Move From Device Control Register Indexed X-form” on page 1055 in Book III for more information on this instruction.

**Special Registers Altered:**
- Implementation-dependent
Chapter 4. Floating-Point Facility [Category: Floating-Point]

4.1 Floating-Point Facility Overview

This chapter describes the registers and instructions that make up the Floating-Point Facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, “IEEE Standard for Binary Floating-Point Arithmetic” (hereafter referred to as “the IEEE standard”). That standard defines certain required "operations" (addition, subtraction, etc.). Herein, the term “floating-point operation" is used to refer to one of these required operations and to additional operations defined (e.g., those performed by Multiply-Add or Reciprocal Estimate instructions). A Non-IEEE mode is also provided. This mode, which may produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

These instructions are divided into two categories:

- computational instructions
  
The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.6 through 4.6.8.

- non-computational instructions
  
The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.2 through 4.6.5, and 4.6.10.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, $\pm\infty$, and values that are “Not a Number” (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to the Floating-Point Facility: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - Infinity--Infinity (VXSI)
  - Infinity--Infinity (VXID)
  - Zero--Zero (VXZDZ)
  - Infinity--Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)
Each floating-point exception, and each category of invalid operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register” on page 116 for a description of these exception and enable bits, and Section 4.4, “Floating-Point Exceptions” on page 124 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Facility Registers

4.2.1 Floating-Point Registers

Implementations of this architecture provide 32 floating-point registers (FPRs). The floating-point instruction formats provide 5-bit fields for specifying the FPRs to be used in the execution of the instruction. The FPRs are numbered 0-31. See Figure 50 on page 116.

Each FPR contains 64 bits that support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the Move and Select instructions, operate on data located in FPRs and, with the exception of the Compare instructions, place the result value into an FPR and optionally (when Rc=1) place status information into the Condition Register. Instruction forms with Rc=1 are part of Category: Floating-Point.Record.

Load Double and Store Double instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. Load Single instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. Store Single instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Instructions are provided that manipulate the Floating-Point Status and Control Register and the Condition Register explicitly. Some of these instructions copy data from an FPR to the Floating-Point Status and Control Register or vice versa.

The computational instructions and the Select instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

4.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 32:55 are status bits. Bits 56:63 are control bits.

The exception bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, or \texttt{mtfsb0} instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be “exception bits”, and only FX is sticky.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

Hypothetical example table:

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
</tbody>
</table>

Every floating-point instruction, except \texttt{mtfsfi} and \texttt{mtfsf}, implicitly sets FPSCR\textsubscript{FX} to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. \texttt{mcrfs}, \texttt{mtfsfi}, \texttt{mtfsf}, \texttt{mtfsb0}, and \texttt{mtfsb1} can alter FPSCR\textsubscript{FX} explicitly.
Floating-Point Enabled Exception Summary (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. `mcrfs`, `mtfsfi`, `mtfsf`, `mtfsb0`, and `mtfsb1` cannot alter FPSCR_FEX explicitly.

Floating-Point Invalid Operation Exception Summary (VX)
This bit is the OR of all the Invalid Operation exception bits. `mcrfs`, `mtfsfi`, `mtfsf`, `mtfsb0`, and `mtfsb1` cannot alter FPSCR_VX explicitly.

Floating-Point Overflow Exception (OX)
See Section 4.4.3, “Overflow Exception” on page 127.

Floating-Point Underflow Exception (UX)
See Section 4.4.4, “Underflow Exception” on page 128.

Floating-Point Zero Divide Exception (ZX)
See Section 4.4.2, “Zero Divide Exception” on page 126.

Floating-Point Inexact Exception (XX)
See Section 4.4.5, “Inexact Exception” on page 128.

Floating-Point Fraction Rounded (FR)
The last Arithmetic or Rounding and Conversion instruction incremented the fraction during rounding. See Section 4.3.6, “Rounding” on page 123. This bit is not sticky.

Floating-Point Fraction Inexact (FI)
The last Arithmetic or Rounding and Conversion instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6. This bit is not sticky.

Floating-Point Result Flags (FPRF)
Arithmetic, rounding, and Convert From Integer instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. Floating-point Compare instructions set this field based on the relative values of the operands being compared. For Convert To Integer instructions, the value placed into FPRF is undefined. Additional details are given below.

Floating-Point Result Class Descriptor (C)
Arithmetic, rounding, and Convert From Integer instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 52 on page 119.

Floating-Point Condition Code (FPCC)
Floating-point Compare instructions set one of

Programming Note

A single-precision operation that produces a denormalized result sets FPRF to indicate a denormalized number. When possible, single-precision denormalized numbers are represented in normalized double format in the target register.

Floating-Point Invalid Operation Exception ($\infty \div \infty$) (VXIDI)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (0 \cdot 0) (VXZDZ)
See Section 4.4.1.

Floating-Point Invalid Operation Exception ($\infty \cdot 0$) (VXIMZ)
See Section 4.4.1.

Floating-Point Invalid Operation Exception (Invalid Compare) (VXVC)
See Section 4.4.1.

Floating-Point Fraction Rounded (FR)
The last Arithmetic or Rounding and Conversion instruction incremented the fraction during rounding. See Section 4.3.6, “Rounding” on page 123. This bit is not sticky.

Floating-Point Fraction Inexact (FI)
The last Arithmetic or Rounding and Conversion instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6. This bit is not sticky.

See the definition of FPSCR_VX, above, regarding the relationship between FPSCR_FI and FPSCR_VX.

Floating-Point Result Flags (FPRF)
Arithmetic, rounding, and Convert From Integer instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. Floating-point Compare instructions set this field based on the relative values of the operands being compared. For Convert To Integer instructions, the value placed into FPRF is undefined. Additional details are given below.

Floating-Point Result Class Descriptor (C)
Arithmetic, rounding, and Convert From Integer instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 52 on page 119.

Floating-Point Condition Code (FPCC)
Floating-point Compare instructions set one of

Programming Note

A single-precision operation that produces a denormalized result sets FPRF to indicate a denormalized number. When possible, single-precision denormalized numbers are represented in normalized double format in the target register.
the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and Convert From Integer instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 52 on page 119. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

**Floating-Point Less Than or Negative** (FL or \(<\))

**Floating-Point Greater Than or Positive** (FG or \(>\))

**Floating-Point Equal or Zero** (FE or \(=\))

**Floating-Point Unordered or NaN** (FU or \(?\))

**Floating-Point Invalid Operation Exception (Software-Defined Condition)** (VXSOFT)

This bit can be altered only by `mcrfs`, `mtfsfi`, `mtfsf`, `mtfsb0`, or `mtfsb1`. See Section 4.4.1.

**Programming Note**

FPSCR\(_{VXSOFT}\) can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation Exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

**Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT)

See Section 4.4.1.

**Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI)

See Section 4.4.1.

**Floating-Point Invalid Operation Exception Enable** (VE)

See Section 4.4.1.

**Floating-Point Overflow Exception Enable** (OE)

See Section 4.4.3, “Overflow Exception” on page 127.

**Floating-Point Underflow Exception Enable** (UE)

See Section 4.4.4, “Underflow Exception” on page 128.

**Floating-Point Zero Divide Exception Enable** (ZE)

See Section 4.4.2, “Zero Divide Exception” on page 126.

**Floating-Point Inexact Exception Enable** (XE)

See Section 4.4.5, “Inexact Exception” on page 128.

**Floating-Point Non-IEEE Mode** (NI)

Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply.

If floating-point non-IEEE mode is implemented, this bit has the following meaning.

0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard).

1 The processor is in floating-point non-IEEE mode.

When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits may have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with FPSCR\(_{NI}=1\), and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode may vary between implementations, and between different executions on the same implementation.

**Programming Note**

When the processor is in floating-point non-IEEE mode, the results of floating-point operations may be approximate, and performance for these operations may be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation may return 0 instead of a denormalized number, and may return a large number instead of an infinity.
4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 53. Floating-point single format

<table>
<thead>
<tr>
<th>S</th>
<th>EXP</th>
<th>FRACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 54. Floating-point double format

Values in floating-point format are composed of three fields:

- **S**: sign bit
- **EXP**: exponent + bias
- **FRACTION**: fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (i.e., the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 55.

```
<table>
<thead>
<tr>
<th>Format</th>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent Bias</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>Maximum Exponent</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>Minimum Exponent</td>
<td>-126</td>
<td>-1022</td>
</tr>
<tr>
<td>Widths (bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Sign</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Exponent</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>Fraction</td>
<td>23</td>
<td>52</td>
</tr>
<tr>
<td>Significand</td>
<td>24</td>
<td>53</td>
</tr>
</tbody>
</table>
```

Figure 55. IEEE floating-point fields

The architecture requires that the FPRs of the Floating-Point Facility support the floating-point double format only.

4.3.2 Value Representation

This architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 56.

```
-Inf -NOR -Den 0 +Den +NOR +Inf
```

Figure 56. Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

**Binary floating-point numbers**

Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.
Normalized numbers (± NOR)
These are values that have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

\[ \text{NOR} = (-1)^s \times 2^E \times (1.\text{fraction}) \]

where \( s \) is the sign, \( E \) is the unbiased exponent, and 1.\text{fraction} is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude \( M \) of a normalized floating-point number are approximately equal to:

Single Format:
\[ 1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38} \]

Double Format:
\[ 2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308} \]

Zero values (± 0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to −0).

Denormalized numbers (± DEN)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

\[ \text{DEN} = (-1)^s \times 2^{E_{\text{min}}} \times (0.\text{fraction}) \]

where \( E_{\text{min}} \) is the minimum representable exponent value (−126 for single-precision, −1022 for double-precision).

Infinities (± ∞)
These are values that have the maximum biased exponent value:

255 in single format
2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

\[-\infty < \text{every finite number} < +\infty\]

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.4.1, "Invalid Operation Exception" on page 126.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

Not a Numbers (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (i.e., NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a Signaling NaN; otherwise it is a Quiet NaN.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions. Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled (FPSCR.VE=0). Quiet NaNs propagate through all floating-point operations except ordered comparison, Floating Round to Single-Precision, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

\[
\text{if (FRA) is a NaN then } FRT \leftarrow \text{(FRA)} \\
\text{else if (FRB) is a NaN then } FRT \leftarrow \text{(FRB)} \\
\text{else if generated QNaN then } FRT \leftarrow \text{generated QNaN}
\]

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is \text{frsp}. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled Invalid Operation Exception.
A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN’s fraction are zero.

**4.3.3 Sign of Result**

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation \(x - y\) is the same as the sign of the result of the add operation \(x + (-y)\).

- When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward \(-\infty\), in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

- The sign of the result of a Square Root or Reciprocal Square Root Estimate operation is always positive, except that the square root of \(-0\) is \(-0\) and the reciprocal square root of \(-0\) is \(-\infty\).

- The sign of the result of a Round to Single-Precision, or Convert From Integer, or Round to Integer operation is the sign of the operand being converted.

For the Multiply-Add instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

**4.3.4 Normalization and Denormalization**

The intermediate result of an arithmetic or \texttt{frsp} instruction may require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 4.5.1, “Execution Model for IEEE Operations” on page 129) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 4.4.4, “Underflow Exception”. These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format’s minimum value. If any significant bits are lost in this shifting process then “Loss of Accuracy” has occurred (See Section 4.4.4, “Underflow Exception” on page 128) and Underflow Exception is signaled.

**4.3.5 Data Handling and Precision**

Most of the Floating-Point Facility Architecture, including all computational, Move, and Select instructions, use the floating-point double format to represent data in the FPRs. Single-precision and integer-valued operands may be manipulated using double-precision operations. Instructions are provided to coerce these values from a double format operand. Instructions are also provided for manipulations which do not require double-precision. In addition, instructions are provided to access a true single-precision representation in storage, and a fixed-point integer representation in GPRs.

**4.3.5.1 Single-Precision Operands**

For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are caused by these instructions. An instruction is provided to explicitly convert a double format operand in an FPR to single-precision. Floating-point single-precision is enabled with four types of instruction.

1. Load Floating-Point Single

This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.
2. Round to Floating-Point Single-Precision

The Floating Round to Single-Precision instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR in double format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the Floating Round to Single-Precision instruction, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

If any input value is not representable in single format and either OE=1 or UE=1, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

For fres[,.] or frsqrtes[,], if the input value is finite and has an unbiased exponent greater than +127, the input value is interpreted as an Infinity.

4. Store Floating-Point Single

This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a Load Floating-Point Single, Floating Round to Single-Precision, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

Programming Note

The Floating Round to Single-Precision instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by fcfid) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a Floating Round to Single-Precision instruction.

Programming Note

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

4.3.5.2 Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and fixed-Point facilities, instructions are provided to convert between floating-point double format and fixed-point integer format in an FPR. Computation on integer-valued operands may be performed using arithmetic instructions of the required precision. (The results may not be integer values.) The two groups of instructions provided specifically to support integer-valued operands are described below.

1. Floating Round to Integer

The Floating Round to Integer instructions round a double-precision operand to an integer value in floating-point double format. These instructions may cause Invalid Operation (VXSNAN) exceptions. See Sections 4.3.6 and 4.5.1 for more information about rounding.

2. Floating Convert To/From Integer

The Floating Convert To/Integer instructions convert a double-precision operand to a 32-bit or 64-bit signed fixed-point integer format. Variants are provided both to perform rounding based on
the value of FPSCR\textsubscript{RN} and to round toward zero. These instructions may cause Invalid Operation (VXSNaN, VXCVI) and Inexact exceptions. The
Floating Convert From Integer instruction converts a 64-bit signed fixed-point integer to a double-precision floating-point integer. Because of the limitations of the source format, only an Inexact exception may be generated.

### 4.3.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 4.3.2, “Value Representation” and Section 4.4, “Floating-Point Exceptions” for the cases not covered here.

The Arithmetic and Rounding and Conversion instructions round their intermediate results. With the exception of the Estimate instructions, these instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target FPR in double format. The Floating Round to Integer and Floating Convert To Integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for Floating Round to Integer is normalized and put in double format, and for Floating Convert To Integer is converted to a signed fixed-point integer.

FPSCR bits FR and FI generally indicate the results of rounding. Each of the instructions which rounds its intermediate result sets these bits. If the fraction is incremented during rounding then FR is set to 1, otherwise FR is set to 0. If the result is inexact then FI is set to 1, otherwise FI is set to zero. The Round to Integer instructions are exceptions to this rule, setting FR and FI to 0. The Estimate instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register”. These are encoded as follows.

<table>
<thead>
<tr>
<th>RN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to Nearest</td>
</tr>
<tr>
<td>01</td>
<td>Round toward Zero</td>
</tr>
<tr>
<td>10</td>
<td>Round toward +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round toward -Infinity</td>
</tr>
</tbody>
</table>

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 57 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. “LSB” means “least significant bit”.

#### Figure 57. Selection of Z1 and Z2

**Round to Nearest**

Choose the value that is closer to Z (Z1 or Z2).

In case of a tie, choose the one that is even (least significant bit 0).

**Round toward Zero**

Choose the smaller in magnitude (Z1 or Z2).

**Round toward +Infinity**

Choose Z1.

**Round toward -Infinity**

Choose Z2.

See Section 4.5.1, “Execution Model for IEEE Operations” on page 129 for a detailed explanation of rounding.
4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity-Infinity
  - Infinity-Infinity
  - Zero/Zero
  - Infinity-Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions, other than Invalid Operation Exception due to Software-Defined Condition, may occur during execution of computational instructions. An Invalid Operation Exception due to Software-Defined Condition occurs when a Move To FPSCR instruction sets FPSCR\_VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 125), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than mtfsi or mtfs, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) is set with Invalid Operation Exception (\(<\times0\)) for Multiply-Add instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for Compare Ordered instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for Convert To Integer instructions.

When an exception occurs the writing of a result to the target register may be suppressed or a result may be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of "traps" and "trap handlers". In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the "trap enabled" case; the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the "default result" value specified for the "trap disabled" (or "no trap occurs" or "trap is not implemented") case; the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled float-
ing-point exception occurs. The system floating-point enabled exception error handler is also invoked if a Move To FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1; the Move To FPSCR instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.) The effects of the four possible settings of these bits are as follows.

**FE0 FE1 Description**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Description</th>
</tr>
</thead>
</table>
| 0 | 0 | Ignore Exceptions Mode  
Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked. |
| 0 | 1 | Imprecise Nonrecoverable Mode  
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked. |
| 1 | 0 | Imprecise Recoverable Mode  
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been affected by or have affected subsequent instructions that are executed before the error handler is invoked. |
| 1 | 1 | Precise Mode  
The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception. |

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction and there is only one such instruction. Otherwise it has not begun execution (or may have been partially executed in some cases, as described in Book III).

**Programming Note**

In any of the three non-Precise modes, a Floating-Point Status and Control Register instruction can be used to force any exceptions, due to instructions initiated before the Floating-Point Status and Control Register instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the Floating-Point Status and Control Register instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. (It always applies in the latter case.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.
4.4.1 Invalid Operation Exception

4.4.1.1 Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a Signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities (\(\infty - \infty\))
- Division of infinity by infinity (\(\infty / \infty\))
- Division of zero by zero (0 ÷ 0)
- Multiplication of infinity by zero (\(\infty \times 0\))
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) number (Invalid Square Root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

An Invalid Operation Exception also occurs when an \(\text{mtfsfi}\), \(\text{mtsf}\), or \(\text{mtfsb1}\) instruction is executed that sets FPSCR\(_{VXSOFT}\) to 1 (Software-Defined Condition).

4.4.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled (FPSCR\(_{VE}\)=1) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - FPSCR\(_{VXS}\) (if SNaN)
   - FPSCR\(_{VX}\) (if \(\infty - \infty\))
   - FPSCR\(_{VX}\) (if \(\infty / \infty\))
   - FPSCR\(_{VX}\) (if 0 ÷ 0)
   - FPSCR\(_{VX}\) (if invalid comp)
   - FPSCR\(_{VXS}\) (if invalid sqrt)
   - FPSCR\(_{VX}\) (if invalid int cvrt)

2. If the operation is an arithmetic, Floating Round to Single-Precision operation, the target FPR is set to a Quiet NaN
   - FPSCR\(_{FRF}\) are set to zero
   - FPSCR\(_{FFP}\) is set to indicate the class of the result (Quiet NaN)

3. If the operation is a convert to 64-bit integer operation, the target FPR is set as follows:
   - FPSCR\(_{FRF}\) are set to zero
   - FPSCR\(_{FFP}\) is undefined

4. If the operation is a compare, FPSCR\(_{FRF}\) are unchanged
   - FPSCR\(_{FFP}\) is set to reflect unordered

5. If an \(\text{mtfsfi}\), \(\text{mtsf}\), or \(\text{mtfsb1}\) instruction is executed that sets FPSCR\(_{VXS}\) to 1, the FPSCR is set as specified in the instruction description.

When Invalid Operation Exception is disabled (FPSCR\(_{VE}\)=0) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - FPSCR\(_{VXS}\) (if SNaN)
   - FPSCR\(_{VX}\) (if \(\infty - \infty\))
   - FPSCR\(_{VX}\) (if \(\infty / \infty\))
   - FPSCR\(_{VX}\) (if 0 ÷ 0)
   - FPSCR\(_{VX}\) (if invalid comp)
   - FPSCR\(_{VXS}\) (if invalid sqrt)
   - FPSCR\(_{VX}\) (if invalid int cvrt)

2. If the operation is an arithmetic or Floating Round to Single-Precision operation, the target FPR is set to a Quiet NaN
   - FPSCR\(_{FRF}\) are set to zero
   - FPSCR\(_{FFP}\) is set to indicate the class of the result (Quiet NaN)

4. If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
   - FPSCR\(_{FRF}\) are set to zero
   - FPSCR\(_{FFP}\) is undefined

5. If the operation is a compare, FPSCR\(_{FRF}\) are unchanged
   - FPSCR\(_{FFP}\) is set to reflect unordered

6. If an \(\text{mtfsfi}\), \(\text{mtsf}\), or \(\text{mtfsb1}\) instruction is executed that sets FPSCR\(_{VXS}\) to 1, the FPSCR is set as specified in the instruction description.

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a Reciprocal Estimate instruction \((\text{fr[s]}\) or \(\text{frsqrte[s]}\)) is executed with an operand value of zero.


4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (FPSCRZE=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   FPSCRZX ← 1
2. The target FPR is unchanged
3. FPSCRFR FI are set to zero
4. FPSCRFPRF is unchanged

When Zero Divide Exception is disabled (FPSCRZE=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   FPSCRZX ← 1
2. The target FPR is set to ±Infinity, where the sign is determined by the XOR of the signs of the operands
3. FPSCRFR FI are set to zero
4. FPSCRFPRF is set to indicate the class and sign of the result (±Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

An Overflow Exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled (FPSCROE=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   FPSCRXX ← 1
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536
3. For single-precision arithmetic instructions and the Floating Round to Single-Precision instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192
4. The adjusted rounded result is placed into the target FPR
5. FPSCRFPRF is set to indicate the class and sign of the result (±Infinity or ±Normal Number)

1. Overflow Exception is set
   FPSCRXX ← 1
2. Inexact Exception is set
   FPSCRXX ← 1
3. The result is determined by the rounding mode (FPSCRRN) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store ±Infinity, where the sign is the sign of the intermediate result
   - Round toward Zero
     Store the format’s largest finite number with the sign of the intermediate result
   - Round toward +Infinity
     For negative overflow, store the format’s most negative finite number; for positive overflow, store +Infinity
   - Round toward −Infinity
     For negative overflow, store −Infinity; for positive overflow, store the format’s largest finite number
4. The result is placed into the target FPR
5. FPSCRF is undefined
6. FPSCRFI is set to 1
7. FPSCRFPRF is set to indicate the class and sign of the result (±Infinity or ±Normal Number)
4.4.4 Underflow Exception

4.4.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- **Enabled:**
  Underflow occurs when the intermediate result is “Tiny”.

- **Disabled:**
  Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A “Tiny” result is detected before rounding, when a non-zero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is “Tiny” and Underflow Exception is disabled (FPSCR\_UE=0) then the intermediate result is denormalized (see Section 4.3.4, “Normalization and Denormalization” on page 121) and rounded (see Section 4.3.6, “Rounding” on page 123) before being placed into the target FPR.

“Loss of Accuracy” is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled (FPSCR\_UE=1) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   \[ FPSCR\_UX \leftarrow 1 \]

2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536

3. For single-precision arithmetic instructions and the Floating Round to Single-Precision instruction, the exponent of the normalized intermediate result is adjusted by adding 192

4. The adjusted rounded result is placed into the target FPR

5. FPSCR\_FPREF is set to indicate the class and sign of the result (± Normalized Number)

When Underflow Exception is disabled (FPSCR\_UE=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   \[ FPSCR\_UX \leftarrow 1 \]

2. The rounded result is placed into the target FPR

3. FPSCR\_FPREF is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)

2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When an Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   \[ FPSCR\_XX \leftarrow 1 \]

2. The rounded or overflowed result is placed into the target FPR

3. FPSCR\_FPREF is set to indicate the class and sign of the result

---

**Programming Note**

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled (FPSCR\_UE=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   \[ FPSCR\_UX \leftarrow 1 \]

2. The rounded result is placed into the target FPR

3. FPSCR\_FPREF is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

---

**Programming Note**

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.
4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 4.3 and Section 4.4 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. The Power ISA follows these guidelines; double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and `fcf` produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>L</th>
<th>FRACTION</th>
<th>G</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 58. IEEE 64-bit execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the shifts with zeros being shifted into the R bit. Figure 59 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

<table>
<thead>
<tr>
<th>G</th>
<th>R</th>
<th>X</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IR is exact</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>IR closer to NL</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>IR midway between NL and NH</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>IR midway between NL and NH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>IR closer to NL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>IR is exact</td>
</tr>
</tbody>
</table>

Figure 59. Interpretation of G, R, and X bits

Figure 60 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figure 58.

<table>
<thead>
<tr>
<th>Format</th>
<th>Guard</th>
<th>Round</th>
<th>Sticky</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>G bit</td>
<td>R bit</td>
<td>X bit</td>
</tr>
</tbody>
</table>

Figure 60. Location of the Guard, Round, and Sticky bits in the IEEE execution model
The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction. Four user-selectable rounding modes are provided through FPSCR_RN as described in Section 4.3.6, “Rounding” on page 123. Using Z1 and Z2 as defined on page 123, the rules for rounding in each mode are as follows.

- **Round to Nearest**
  - **Guard bit = 0**
    The result is truncated. (Result exact (GRX=000) or closest to next lower value in magnitude (GRX=001, 010, or 011))
  - **Guard bit = 1**
    Depends on Round and Sticky bits:
    - **Case a**
      If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX=101, 110, or 111))
    - **Case b**
      If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).

- **Round toward Zero**
  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.

- **Round toward + Infinity**
  Choose Z1.

- **Round toward - Infinity**
  Choose Z2.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. If any of the Guard, Round, or Sticky bits is nonzero, then the result is also inexact. Fraction bits are stored to the target FPR. For Floating Round to Integer, Floating Round to Single-Precision, and single-precision arithmetic instructions, low-order zeros must be appended as appropriate to fill out the double-precision fraction.
4.5.2 Execution Model for Multiply-Add Type Instructions

The Power ISA provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

![Figure 61. Multiply-add 64-bit execution model](image)

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 62 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

![Figure 62. Location of the Guard, Round, and Sticky bits in the multiply-add execution model](image)

The rules for rounding the intermediate result are the same as those given in Section 4.5.1.
4.6 Floating-Point Facility Instructions

For each instruction in this section that defines the use of an Rc bit, the behavior defined for the instruction corresponding to Rc=1 is considered part of the Floating-Point.Record category.
4.6.1 Floating-Point Storage Access Instructions

The Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 26.

Programming Note

The ia extended mnemonic permits computing an effective address as a Load or Store instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. This extended mnemonic is described in Section E.10, “Miscellaneous Mnemonics” on page 719.

4.6.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.

4.6.2 Floating-Point Load Instructions

There are three basic forms of load instruction: single-precision, double-precision, and integer. The integer form is provided by the Load Floating-Point as Integer Word Algebraic instruction, described on page 136. Because the FPRs support only floating-point double format, single-precision Load Floating-Point instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let WORD0:31 be the floating-point single-precision operand accessed from storage.

Normalized Operand

if WORD1:8 > 0 and WORD1:8 < 255 then
  FRT0 ← WORD0:1
  FRT2 ← -WORD1
  FRT3 ← -WORD1
  FRT4 ← -WORD1
  FRT5:63 ← WORD2:31 || 290

Denormalized Operand

if WORD1:8 = 0 and WORD9:31 ≠ 0 then
  sign ← WORD0
  exp ← -126
  frac0:52 ← 0b0 || WORD9:31 || 290
  normalize the operand
  do while frac0 = 0
    frac0:52 ← frac1:52 || 0b0

exp ← exp - 1
FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52

Zero / Infinity / NaN

if WORD1:8 = 255 or WORD1:31 = 0 then
  FRT0:1 ← WORD0:1
  FRT2 ← WORD1
  FRT3 ← WORD1
  FRT4 ← WORD1
  FRT5:63 ← WORD2:31 || 290

For double-precision Load Floating-Point instructions and for the Load Floating-Point as Integer Word Algebraic instruction no conversion is required, as the data from storage are copied directly into the FPR.

Many of the Load Floating-Point instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA ≠ 0, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

Note: Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.
**Load Floating-Point Single D-form**

```
Ifs FRT,D(RA)
```

<table>
<thead>
<tr>
<th>0</th>
<th>48</th>
<th>FRT</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))

Let the effective address (EA) be the sum (RA|0)+D.
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.

**Special Registers Altered:**
None

---

**Load Floating-Point Single Indexed X-form**

```
Ifsx FRT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>FRT</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>RB</th>
<th>535</th>
<th>1</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
FRT ← DOUBLE(MEM(EA, 4))

Let the effective address (EA) be the sum (RA|0)+(RB).
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.

**Special Registers Altered:**
None

---

**Load Floating-Point Single with Update D-form**

```
Ifsu FRT,D(RA)
```

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
<th>FRT</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>D</th>
<th>31</th>
</tr>
</thead>
</table>

EA ← (RA) + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA

Let the effective address (EA) be the sum (RA)+D.
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.
EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None

---

**Load Floating-Point Single with Update Indexed X-form**

```
Ifsux FRT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>FRT</th>
<th>11</th>
<th>RA</th>
<th>16</th>
<th>RB</th>
<th>567</th>
<th>1</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA

Let the effective address (EA) be the sum (RA)+(RB).
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.
EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**
None
Load Floating-Point Double D-form

Ifd  FRT,D(RA)

<table>
<thead>
<tr>
<th>50</th>
<th>FRT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
FRT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0)+D.
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:
None

Load Floating-Point Double Indexed X-form

Ifdx  FRT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>599</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
FRT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0)+(RB).
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:
None

Load Floating-Point Double with Update D-form

Ifdu  FRT,D(RA)

<table>
<thead>
<tr>
<th>51</th>
<th>FRT</th>
<th>RA</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

EA ← (RA) + EXTS(D)
FRT ← MEM(EA, 8)
RA ← EA

Let the effective address (EA) be the sum (RA)+D.
The doubleword in storage addressed by EA is loaded into register FRT.
EA is placed into register RA.
If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double with Update Indexed X-form

Ifdux  FRT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>FRT</th>
<th>RA</th>
<th>RB</th>
<th>631</th>
</tr>
</thead>
</table>

EA ← (RA) + (RB)
FRT ← MEM(EA, 8)
RA ← EA

Let the effective address (EA) be the sum (RA)+(RB).
The doubleword in storage addressed by EA is loaded into register FRT.
EA is placed into register RA.
If RA=0, the instruction form is invalid.

Special Registers Altered:
None
**Load Floating-Point as Integer Word**  
**Algebraic Indexed X-form**

\[ \text{lfiwax } \quad \text{FRT,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>855</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>16</td>
<td>RB</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
FRT ← EXTS(MEM(EA, 4))

Let the effective address (EA) be the sum (RA|0)+(RB).

The word in storage addressed by EA is loaded into FRT\(_{32:63}\). FRT\(_{0:31}\) are filled with a copy of bit 0 of the loaded word.

**Special Registers Altered:**  
None

**Load Floating-Point as Integer Word and Zero Indexed X-form**

\[ \text{lfiwzx } \quad \text{FRT,RA,RB} \]

[Category: Floating-Point.Phased-in]

<table>
<thead>
<tr>
<th>31</th>
<th>887</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>16</td>
<td>RB</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
FRT ← 32\(_0\) || MEM(EA, 4)

Let the effective address (EA) be the sum (RA|0)+(RB).

The word in storage addressed by EA is loaded into FRT\(_{32:63}\). FRT\(_{0:31}\) are set to 0.

**Special Registers Altered:**  
None
### 4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction: single-precision, double-precision, and integer. The integer form is provided by the `Store Floating-Point as Integer Word` instruction, described on page 140. Because the FPRs support only floating-point double format for floating-point data, single-precision `Store Floating-Point` instructions convert double-precision data to single format prior to storing the operand into storage. The conversion steps are as follows.

Let \( \text{WORD}_{0:31} \) be the word in storage written to.

**No Denormalization Required (includes Zero / Infinity / NaN)**

if \( \text{FRS}_{1:11} > 896 \) or \( \text{FRS}_{1:63} = 0 \) then

\[
\begin{align*}
\text{WORD}_{0:1} & \leftarrow \text{FRS}_{0:1} \\
\text{WORD}_{2:31} & \leftarrow \text{FRS}_{5:34}
\end{align*}
\]

**Denormalization Required**

if \( 874 \leq \text{FRS}_{1:11} \leq 896 \) then

\[
\begin{align*}
\text{sign} & \leftarrow \text{FRS}_{0} \\
\text{exp} & \leftarrow \text{FRS}_{1:11} - 1023 \\
\text{frac}_{0:52} & \leftarrow 0b1 || \text{FRS}_{12:63} \\
\text{denormalize operand} & \text{do while exp} < -126 \\
& \quad \text{frac}_{0:52} \leftarrow 0b0 || \text{frac}_{0:51} \\
& \quad \text{exp} \leftarrow \text{exp} + 1 \\
\text{WORD}_{0} & \leftarrow \text{sign} \\
\text{WORD}_{1:8} & \leftarrow 0x00 \\
\text{WORD}_{9:31} & \leftarrow \text{frac}_{1:23}
\end{align*}
\]

else \( \text{WORD} \leftarrow \text{undefined} \)

Notice that if the value to be stored by a single-precision `Store Floating-Point` instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in \( \text{WORD} \) is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision `Load Floating-Point` from \( \text{WORD} \) will not compare equal to the contents of the original source register).

For double-precision `Store Floating-Point` instructions and for the `Store Floating-Point as Integer Word` instruction no conversion is required, as the data from the FPR are copied directly into storage.

Many of the `Store Floating-Point` instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if \( \text{RA} \neq 0 \), the effective address is placed into register RA.

**Note:** Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.
### Store Floating-Point Single D-form

stfs FRS,D(RA)

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))

Let the effective address (EA) be the sum (RA|0)+D.
The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

**Special Registers Altered:**

None

### Store Floating-Point Single Indexed X-form

stfx FRS,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>663</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← SINGLE((FRS))

Let the effective address (EA) be the sum (RA|0)+(RB).
The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

**Special Registers Altered:**

None

### Store Floating-Point Single with Update D-form

stfsu FRS,D(RA)

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA

Let the effective address (EA) be the sum (RA|0)+D.
The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

EA is placed into register RA.
If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None

### Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>695</td>
</tr>
</tbody>
</table>

EA ← (RA) + (RB)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA

Let the effective address (EA) be the sum (RA|0)+(RB).
The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

EA is placed into register RA.
If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None
**Store Floating-Point Double D-form**

```assembly
stfd FRS,D(RA)
```

<table>
<thead>
<tr>
<th>54</th>
<th>FRS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

If RA = 0 then \( b \leftarrow 0 \)

Else \( b \leftarrow (RA) \)

\( EA \leftarrow b + \text{EXTS}(D) \)

\( \text{MEM}(EA, 8) \leftarrow \text{FRS} \)

Let the effective address (EA) be the sum \((RA)+D\).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

**Special Registers Altered:**

None

---

**Store Floating-Point Double Indexed X-form**

```assembly
stfdx FRS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>727</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

If RA = 0 then \( b \leftarrow 0 \)

Else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)

\( \text{MEM}(EA, 8) \leftarrow \text{FRS} \)

Let the effective address (EA) be the sum \((RA)+(RB)\).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

**Special Registers Altered:**

None

---

**Store Floating-Point Double with Update D-form**

```assembly
stfdu FRS,D(RA)
```

<table>
<thead>
<tr>
<th>55</th>
<th>FRS</th>
<th>RA</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\( EA \leftarrow (RA) + \text{EXTS}(D) \)

\( \text{MEM}(EA, 8) \leftarrow \text{FRS} \)

\( RA \leftarrow EA \)

Let the effective address (EA) be the sum \((RA)+D\).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None

---

**Store Floating-Point Double with Update Indexed X-form**

```assembly
stfdux FRS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>759</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

\( EA \leftarrow (RA) + (RB) \)

\( \text{MEM}(EA, 8) \leftarrow \text{FRS} \)

\( RA \leftarrow EA \)

Let the effective address (EA) be the sum \((RA)+(RB)\).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None
Store Floating-Point as Integer Word
Indexed X-form

stfiwx \ FRS,RA,RB

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>983</th>
<th>/</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (FRS)_{32:63}

Let the effective address (EA) be the sum (RA)(0)+(RB).

(FRS)_{32:63} are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a Load Floating-Point Single instruction, a single-precision Arithmetic instruction, or \texttt{frsp}, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more Floating-Point Move instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:
None
4.6.4 Floating-Point Load Store Doubleword Pair Instructions [Category: Floating-Point.Phased-Out]

For \textit{lfdp[x]}, the doubleword-pair in storage addressed by EA is loaded into an even-odd pair of FPRs with the even-numbered FPR being loaded with the leftmost doubleword from storage and the odd-numbered FPR being loaded with the rightmost doubleword.

For \textit{stfdp[x]}, the content of an even-odd pair of FPRs is stored into the doubleword-pair in storage addressed by EA, with the even-numbered FPR being stored into the leftmost doubleword in storage and the odd-numbered FPR being stored into the rightmost doubleword.

---

**Programming Note**

The instructions described in this section should not be used to access an operand in DFP128 format when LE $\neq$ SLE.

---

### Load Floating-Point Double Pair DS-form

\begin{itemize}
  \item \textbf{If} RA = 0 then b $\leftarrow$ 0
  \item else b $\leftarrow$ (RA)
  \item EA $\leftarrow$ b + EXTS(DS||0b00)
  \item FRTp $\leftarrow$ MEM(EA, 16)
\end{itemize}

Let the effective address (EA) be the sum (RA|0) + (DS||0b00). The doubleword-pair in storage addressed by EA is placed into register-pair FRTp.

If FRTp is odd, the instruction form is invalid.

**Special Registers Altered:**

None

---

### Store Floating-Point Double Pair DS-form

\begin{itemize}
  \item if RA = 0 then b $\leftarrow$ 0
  \item else b $\leftarrow$ (RA)
  \item EA $\leftarrow$ b + EXTS(DS||0b00)
  \item MEM(EA, 16) $\leftarrow$ FRSp
\end{itemize}

Let the effective address (EA) be the sum (RA|0) + (DS||0b00). The contents of register-pair FRSp are stored into the doubleword-pair in storage addressed by EA.

If FRSp is odd, the instruction form is invalid.

**Special Registers Altered:**

None

---

### Load Floating-Point Double Pair Indexed X-form

\begin{itemize}
  \item if RA = 0 then b $\leftarrow$ 0
  \item else b $\leftarrow$ (RA)
  \item EA $\leftarrow$ b + (RB)
  \item FRTp $\leftarrow$ MEM(EA, 16)
\end{itemize}

Let the effective address (EA) be the sum (RA|0) + (RB). The doubleword-pair in storage addressed by EA is placed into register-pair FRTp.

If FRTp is odd, the instruction form is invalid.

**Special Registers Altered:**

None

---

### Store Floating-Point Double Pair Indexed X-form

\begin{itemize}
  \item if RA = 0 then b $\leftarrow$ 0
  \item else b $\leftarrow$ (RA)
  \item EA $\leftarrow$ b + (RB)
  \item MEM(EA, 16) $\leftarrow$ FRSp
\end{itemize}

Let the effective address (EA) be the sum (RA|0) + (RB). The contents of register-pair FRSp are stored into the doubleword-pair in storage addressed by EA.

If FRSp is odd, the instruction form is invalid.

**Special Registers Altered:**

None
4.6.5 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for \texttt{fneg}, \texttt{fabs}, \texttt{fnabs}, and \texttt{fcpsgn}. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by \texttt{fneg}, \texttt{fabs}, \texttt{fnabs}, and \texttt{fcpsgn}). These instructions do not alter the FPSCR.

### Floating Move Register X-form

\begin{verbatim}
Floating Move Register X-form
fmr      FRT,FRB \\
fmr.     FRT,FRB \\
\end{verbatim}

The contents of register FRB are placed into register FRT.

**Special Registers Altered:**

- CR1 (if Rc=1)

### Floating Negate X-form

\begin{verbatim}
Floating Negate X-form
fneg     FRT,FRB \\
fneg.    FRT,FRB \\
\end{verbatim}

The contents of register FRB with bit 0 inverted are placed into register FRT.

**Special Registers Altered:**

- CR1 (if Rc=1)

### Floating Absolute Value X-form

\begin{verbatim}
Floating Absolute Value X-form
fabs     FRT,FRB \\
fabs.   FRT,FRB \\
\end{verbatim}

The contents of register FRB with bit 0 set to zero are placed into register FRT.

**Special Registers Altered:**

- CR1 (if Rc=1)

### Floating Copy Sign X-form

\begin{verbatim}
Floating Copy Sign X-form
fcpsgn   FRT,FRA,FRB \\
fcpsgn.  FRT,FRA,FRB \\
\end{verbatim}

The contents of register FRB with bit 0 set to the value of bit 0 of register FRA are placed into register FRT.

**Special Registers Altered:**

- CR1 (if Rc=1)

### Floating Negative Absolute Value X-form

\begin{verbatim}
Floating Negative Absolute Value X-form
fnabs    FRT,FRB \\
fnabs.   FRT,FRB \\
\end{verbatim}

The contents of register FRB with bit 0 set to one are placed into register FRT.

**Special Registers Altered:**

- CR1 (if Rc=1)
Floating Merge Even Word X-form
[Category: Vector-Scalar]

fmrgew FRT,FRA,FRB

if MSR.FP=0 then FP_Unavailable()  
FPR[FRT].word[0] ← FPR[FRA].word[0]  
FPR[FRT].word[1] ← FPR[FRB].word[0]  

The contents of word element 0 of FPR[FRA] are placed into word element 0 of FPR[FRT].  
The contents of word element 0 of FPR[FRB] are placed into word element 1 of FPR[FRT].  

fmrgew is treated as a Floating-Point instruction in terms of resource availability.

Special Registers Altered
None

Floating Merge Odd Word X-form
[Category: Vector-Scalar]

fmrgow FRT,FRA,FRB

if MSR.FP=0 then FP_Unavailable()  
FPR[FRT].word[0] ← FPR[FRA].word[1]  

The contents of word element 1 of FPR[FRA] are placed into word element 0 of FPR[FRT].  
The contents of word element 1 of FPR[FRB] are placed into word element 1 of FPR[FRT].  

fmrgow is treated as a Floating-Point instruction in terms of resource availability.

Special Registers Altered
None
4.6.6 Floating-Point Arithmetic Instructions

4.6.6.1 Floating-Point Elementary Arithmetic Instructions

**Floating Add [Single] A-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadd</td>
<td>FRT,FRA,FRB (Rc=0)</td>
<td>Floating Add [Single] A-form</td>
</tr>
<tr>
<td>fadd.</td>
<td>FRT,FRA,FRB (Rc=1)</td>
<td>Floating Add [Single] A-form</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is added to the floating-point operand in register FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum’s significand is shifted right one bit position and the exponent is increased by one.

FPSCR<sub>FPRF</sub> is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR<sub>VE</sub>=1.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX XX
- VXSNAN VXISI
- CR1 (if Rc=1)

---

**Floating Subtract [Single] A-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsub</td>
<td>FRT,FRA,FRB (Rc=0)</td>
<td>Floating Subtract [Single] A-form</td>
</tr>
<tr>
<td>fsub.</td>
<td>FRT,FRA,FRB (Rc=1)</td>
<td>Floating Subtract [Single] A-form</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

FPSCR<sub>FPRF</sub> is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR<sub>VE</sub>=1.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX XX
- VXSNAN VXISI
- CR1 (if Rc=1)
### Floating Multiply [Single] A-form

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
<th>Rounding Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmul</td>
<td>FRT,FRA,FRC</td>
<td>Rc=0</td>
</tr>
<tr>
<td>fmul.</td>
<td>FRT,FRA,FRC</td>
<td>Rc=1</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR<sub>FPRF</sub> is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR<sub>VE</sub>=1.

**Special Registers Altered:**
- FPRF  FR  FI
- FX  OX  UX  XX
- VXSNAN  VXIMZ
- CR1 (if Rc=1)

### Floating Divide [Single] A-form

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
<th>Rounding Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv</td>
<td>FRT,FRA,FRB</td>
<td>Rc=0</td>
</tr>
<tr>
<td>fdiv.</td>
<td>FRT,FRA,FRB</td>
<td>Rc=1</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR<sub>FPRF</sub> is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR<sub>VE</sub>=1 and Zero Divide Exceptions when FPSCR<sub>ZE</sub>=1.

**Special Registers Altered:**
- FPRF  FR  FI
- FX  OX  UX  ZX  XX
- VXSNAN  VXIDI  VXZDZ
- CR1 (if Rc=1)
### Floating Square Root [Single] A-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
<th>(Rc)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fsqrt</code></td>
<td>FRT, FRB</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td><code>fsqrt</code></td>
<td>FRT, FRB</td>
<td>(Rc=1)</td>
<td></td>
</tr>
<tr>
<td><code>fsqrts</code></td>
<td>FRT, FRB</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td><code>fsqrts</code></td>
<td>FRT, FRB</td>
<td>(Rc=1)</td>
<td></td>
</tr>
</tbody>
</table>

The square root of the floating-point operand in register FRB is placed into register FRT.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>(-0)</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>(+0)</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>QNaN</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\) No result if FPSCR\(_{VE}\) = 1

FPSCR\(_{FPRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE}\)=1.

#### Special Registers Altered:

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
<th>VXSQAN</th>
<th>VXSQRT</th>
<th>CR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Floating Reciprocal Estimate [Single] A-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
<th>(Rc)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fre</code></td>
<td>FRT, FRB</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td><code>fre</code></td>
<td>FRT, FRB</td>
<td>(Rc=1)</td>
<td></td>
</tr>
<tr>
<td><code>fres</code></td>
<td>FRT, FRB</td>
<td>(Rc=0)</td>
<td></td>
</tr>
<tr>
<td><code>fres</code></td>
<td>FRT, FRB</td>
<td>(Rc=1)</td>
<td></td>
</tr>
</tbody>
</table>

An estimate of the reciprocal of the floating-point operand in register FRB is placed into register FRT.

Unless the reciprocal would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate is correct to a precision of one part in \(256\) of the reciprocal of \((FRB)\), i.e.,

\[
\text{ABS}(\text{estimate} - \frac{1}{x}) \leq \frac{1}{256}
\]

where \(x\) is the initial value in FRB.

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>None</td>
</tr>
<tr>
<td>(-0)</td>
<td>(-\infty)</td>
<td>ZX</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+\infty)</td>
<td>ZX</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(+0)</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^2)</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

\(^1\) No result if FPSCR\(_{VE}\) = 1.

\(^2\) No result if FPSCR\(_{ZE}\) = 1.

FPSCR\(_{FPRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\(_{VE}\)=1 and Zero Divide Exceptions when FPSCR\(_{ZE}\)=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

#### Special Registers Altered:

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR (undefined)</th>
<th>FI (undefined)</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX (undefined)</th>
<th>VXSQAN</th>
<th>CR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>(if Rc=1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Floating Reciprocal Square Root Estimate

[Single] A-form

frsqrte FRT,FRB (Rc=0)
frsqrte FRT,FRB (Rc=1)
frsqrtes FRT,FRB (Rc=0)
frsqrtes FRT,FRB (Rc=1)

A estimate of the reciprocal of the square root of the floating-point operand in register FRB is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 32 of the reciprocal of the square root of (FRB), i.e.,

\[ \text{ABS}(\text{estimate} - 1/(\sqrt{x})) \leq \frac{1}{32} \]

where \( x \) is the initial value in FRB.

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>QNan²</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>QNan²</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>-0</td>
<td>-∞ (^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>+0</td>
<td>+∞ (^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>±∞</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNan²</td>
<td>VXSQAN</td>
</tr>
<tr>
<td>QNan</td>
<td>QNan</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if FPSCRZE = 1.
2. No result if FPSCRVE = 1.

FPSCR\(_{FPRF}\) is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCRVE=1 and Zero Divide Exceptions when FPSCRZE=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

- FPRF FR (undefined) FI (undefined)
- FX ZX XX (undefined)
- VXSQAN VXSQRT
- CR1 (if Rc=1)

Note

See the Notes that appear with \text{fre}s.
Floating Test for software Divide X-form

ftdiv BF,FRA,FRB

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>///</th>
<th>FRA</th>
<th>FRB</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

Let $e_a$ be the unbiased exponent of the double-precision floating-point operand in register FRA.

Let $e_b$ be the unbiased exponent of the double-precision floating-point operand in register FRB.

$fe\_flag$ is set to 1 if any of the following conditions occurs.

- The double-precision floating-point operand in register FRA is a NaN or an Infinity.
- The double-precision floating-point operand in register FRB is a Zero, a NaN, or an Infinity.
- $e_b$ is less than or equal to -1022.
- $e_b$ is greater than or equal to 1021.
- The double-precision floating-point operand in register FRA is not a zero and the difference, $e_a - e_b$, is greater than or equal to 1023.
- The double-precision floating-point operand in register FRA is not a zero and the difference, $e_a - e_b$, is less than or equal to -1021.
- The double-precision floating-point operand in register FRA is not a zero and $e_a$ is less than or equal to -970.

Otherwise $fe\_flag$ is set to 0.

$fg\_flag$ is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register FRA is an Infinity.
- The double-precision floating-point operand in register FRB is a Zero, an Infinity, or a denormalized value.

Otherwise $fg\_flag$ is set to 0.

If the implementation guarantees a relative error of $fre_{\text{rel}}$[.] of less than or equal to $2^{-14}$, then $fl\_flag$ is set to 1. Otherwise $fl\_flag$ is set to 0.

CR field BF is set to the value $fl\_flag || fg\_flag || fe\_flag || 0b0$.

Special Registers Altered:
- CR field BF

Programming Note

$ftdiv$ and $ftsqrt$ are provided to accelerate software emulation of divide and square root operations, by performing the requisite special case checking. Software needs only a single branch, on FE=1 (in CR[BF]), to a special case handler. FG and FL may provide further acceleration opportunities.

Floating Test for software Square Root X-form

ftsqrt BF,FRB

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>///</th>
<th>FRB</th>
<th>160</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

Let $e_b$ be the unbiased exponent of the double-precision floating-point operand in register FRB.

$fe\_flag$ is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register FRB is a zero, a NaN, or an infinity, or a negative value.
- $e_b$ is less than or equal to -970.

Otherwise $fe\_flag$ is set to 0.

$fg\_flag$ is set to 1 if the following condition occurs.

- The double-precision floating-point operand in register FRB is a Zero, an Infinity, or a denormalized value.

Otherwise $fg\_flag$ is set to 0.

If the implementation guarantees a relative error of $frsqrte_{\text{rel}}$[.] of less than or equal to $2^{-14}$, then $fl\_flag$ is set to 1. Otherwise $fl\_flag$ is set to 0.

CR field BF is set to the value $fl\_flag || fg\_flag || fe\_flag || 0b0$.

Special Registers Altered:
- CR field BF
4.6.6.2 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows.
- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (fmu[s], followed by fadd[s] or fsub[s]). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

### Floating Multiply-Add [Single] A-form

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source Registers</th>
<th>Destination Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmadd</td>
<td>FRT,FRA,FRC,FRB</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>fmadd.</td>
<td>FRT,FRA,FRC,FRB</td>
<td>(Rc=1)</td>
</tr>
<tr>
<td>fmul</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
<tr>
<td>fmul.</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
<tr>
<td>fmul.s</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
<tr>
<td>fmul.s.</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
</tbody>
</table>

The operation
\[ FRT \leftarrow ([FRA] \times [FRC]) + (FRB) \]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

FPSCR\_FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\_VE=1.

**Special Registers Altered:**
- FPRF, FR, FI
- FX, OX, UX, XX
- VXSNAN, VXISI, VXIMZ
- CR1 (if Rc=1)

### Floating Multiply-Subtract [Single] A-form

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source Registers</th>
<th>Destination Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmsub</td>
<td>FRT,FRA,FRC,FRB</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>fmsub.</td>
<td>FRT,FRA,FRC,FRB</td>
<td>(Rc=1)</td>
</tr>
<tr>
<td>fmsub</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
<tr>
<td>fmsub.</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
<tr>
<td>fmsub.s</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
<tr>
<td>fmsub.s.</td>
<td>FRT,FRA,FRC,FRB</td>
<td></td>
</tr>
</tbody>
</table>

The operation
\[ FRT \leftarrow ([FRA] \times [FRC]) - (FRB) \]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

FPSCR\_FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\_VE=1.

**Special Registers Altered:**
- FPRF, FR, FI
- FX, OX, UX, XX
- VXSNAN, VXISI, VXIMZ
- CR1 (if Rc=1)
Floating Negative Multiply-Add [Single] A-form

fnmadd FRT,FRA,FRC,FRB (Rc=0)
fnmadd. FRT,FRA,FRC,FRB (Rc=1)

fnmadds FRT,FRA,FRC,FRB (Rc=0)
fnmadds. FRT,FRA,FRC,FRB (Rc=1)

The operation

\[
\text{FRT} \leftarrow - \left( \left( \text{FRA} \times \text{FRC} \right) + \text{FRB} \right)
\]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FPSCR\text{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\text{VE}=1.

Special Registers Altered:

\[
\begin{array}{llllllll}
\text{FPRF} & \text{FR} & \text{FI} & \text{FX} & \text{OX} & \text{UX} & \text{XX} & \text{VXSNAN} & \text{VXISI} & \text{VXIMZ} & \text{CR1} \\
\end{array}
\]

(if Rc=1)

Floating Negative Multiply-Subtract [Single] A-form

fnmsub FRT,FRA,FRC,FRB (Rc=0)
fnmsub. FRT,FRA,FRC,FRB (Rc=1)

fnmsubs FRT,FRA,FRC,FRB (Rc=0)
fnmsubs. FRT,FRA,FRC,FRB (Rc=1)

The operation

\[
\text{FRT} \leftarrow - \left( \left( \text{FRA} \times \text{FRC} \right) - \text{FRB} \right)
\]

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the Floating Multiply-Subtract instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

FPSCR\text{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR\text{VE}=1.

Special Registers Altered:

\[
\begin{array}{llllllll}
\text{FPRF} & \text{FR} & \text{FI} & \text{FX} & \text{OX} & \text{UX} & \text{XX} & \text{VXSNAN} & \text{VXISI} & \text{VXIMZ} & \text{CR1} \\
\end{array}
\]

(if Rc=1)
4.6.7 Floating-Point Rounding and Conversion Instructions

### 4.6.7.1 Floating-Point Rounding Instruction

**Floating Round to Single-Precision X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frsp</td>
<td>FRT,FRB (Rc=0)</td>
</tr>
<tr>
<td>frsp.</td>
<td>FRT,FRB (Rc=1)</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to single-precision, using the rounding mode specified by FPSCRRN, and placed into register FRT.

The rounding is described fully in Section A.1, “Floating-Point Round to Single-Precision Model” on page 685.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{VE}=1.

**Special Registers Altered:**
- FPRF (undefined)
- FR
- FI
- FX
- OX
- UX
- XX
- VXSNAN
- CR1 (if Rc=1)

#### Programming Note

Examples of uses of these instructions to perform various conversions can be found in Section F.2, “Floating-Point Conversions [Category: Floating-Point]” on page 726.

### 4.6.7.2 Floating-Point Convert To/From Integer Instructions

**Floating Convert To Integer Doubleword X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fctid</td>
<td>FRT,FRB (Rc=0)</td>
</tr>
<tr>
<td>fctid.</td>
<td>FRT,FRB (Rc=1)</td>
</tr>
</tbody>
</table>

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by FPSCRRN.

If the rounded value is greater than $2^{63}-1$, then the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than $-2^{63}$, then the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR_{FPRC} is undefined. FPSCR_{FR} is set if the result is incremented when rounded. FPSCR_{FI} is set if the result is inexact.

**Special Registers Altered:**
- FPRF (undefined)
- FR
- FI
- FX
- XX
- VXSNAN
- VXCVI
- CR1 (if Rc=1)
Floating Convert To Integer Doubleword with round toward Zero  X-form

fctidz FRT,FRB (Rc=0)
fctidz. FRT,FRB (Rc=1)

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2^{63}-1, then the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, then the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR_{FPRF} is undefined. FPSCR_{FR} is set if the result is incremented when rounded. FPSCR_{FI} is set if the result is inexact.

Special Registers Altered:
FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Doubleword Unsigned  X-form

fctidu FRT,FRB (Rc=0)
fctidu. FRT,FRB (Rc=1)

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by FPSCR_{RN}.

If the rounded value is greater than 2^{64}-1, then the result is 0xFFFF_FFFF_FFFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000_0000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR_{FPRF} is undefined. FPSCR_{FR} is set if the result is incremented when rounded. FPSCR_{FI} is set if the result is inexact.

Special Registers Altered:
FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)
Floating Convert To Integer Doubleword
Unsigned with round toward Zero X-form

Let \( src \) be the double-precision floating-point value in FRB.

If \( src \) is a NaN, then the result is 0x0000_0000_0000_0000, VXCVI is set to 1, and, if \( src \) is an SNaN, VXSNAN is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2\(^{64}-1\), then the result is 0xFFFF_FFFF_FFFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000_0000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR\(_{FRF}\) is undefined. FPSCR\(_{FR}\) is set if the result is incremented when rounded. FPSCR\(_{FI}\) is set if the result is inexact.

Special Registers Altered:
- FPFRF (undefined) FR FI
- FX XX
- VXSNAN VXCVI
- CR1 (if Rc=1)

---

Floating Convert To Integer Word X-form

fctiw FRT,FRB (Rc=0)
fctiw. FRT,FRB (Rc=1)

Let \( src \) be the double-precision floating-point value in FRB.

If \( src \) is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and, if \( src \) is an SNaN, VXSNAN is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode specified by FPSCR\(_{RN}\).

If the rounded value is greater than 2\(^{31}-1\), then the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2\(^{31}\), then the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT\(_{32:63}\) and FRT\(_{0:31}\) is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR\(_{FRF}\) is undefined. FPSCR\(_{FR}\) is set if the result is incremented when rounded. FPSCR\(_{FI}\) is set if the result is inexact.

Special Registers Altered:
- FPFRF (undefined) FR FI
- FX XX
- VXSNAN VXCVI
- CR1 (if Rc=1)
Floating Convert To Integer Word with round toward Zero X-form

\[ \text{fctiwz} \quad \text{FRT}, \text{FRB} \quad (\text{Rc}=0) \]
\[ \text{fctiwz} \quad \text{FRT}, \text{FRB} \quad (\text{Rc}=1) \]

Let \( \text{src} \) be the double-precision floating-point value in FRB.

If \( \text{src} \) is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and, if \( \text{src} \) is an SNaN, VXSNAN is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than \( 2^{31} \cdot 1 \), then the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than \( -2^{31} \), then the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT_{32:63} and FRT_{0:31} is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR_{FFRF} is undefined. FPSCR_{FR} is set if the result is incremented when rounded. FPSCR_{FI} is set if the result is inexact.

Special Registers Altered:
- FPFRF (undefined) FR FI
- FX XX
- VXSNAN VXCVI
- CR1 (if Rc=1)

Floating Convert To Integer Word Unsigned X-form

\[ \text{fctiwu} \quad \text{FRT}, \text{FRB} \quad (\text{Rc}=0) \]
\[ \text{fctiwu} \quad \text{FRT}, \text{FRB} \quad (\text{Rc}=1) \]

Let \( \text{src} \) be the double-precision floating-point value in FRB.

If \( \text{src} \) is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if \( \text{src} \) is an SNaN, VXSNAN is set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode specified by FPSCR_{RN}.

If the rounded value is greater than \( 2^{32} \cdot 1 \), then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than \( 0.0 \), then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT_{32:63} and FRT_{0:31} is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR_{FFRF} is undefined. FPSCR_{FR} is set if the result is incremented when rounded. FPSCR_{FI} is set if the result is inexact.

Special Registers Altered:
- FPFRF (undefined) FR FI
- FX XX
- VXSNAN VXCVI
- CR1 (if Rc=1)
Floating Convert To Integer Word
Unsigned with round toward Zero X-form

[Category: Floating-Point.Phased-In]

fctiwuz FRT,FRB (Rc=0)
fctiwuz FRT,FRB (Rc=1)

Let \( src \) be the double-precision floating-point value in FRB.

If \( src \) is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if \( src \) is an SNaN, VXSNAN is set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than \( 2^{32}-1 \), then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0.0, then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT 32:63 and FRT 0:31 is undefined.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, FPSCR FPRF is undefined. FPSCR FR is set if the result is incremented when rounded. FPSCR FI is set if the result is inexact.

Special Registers Altered:
FPRF FR FI
FX XX
VXSNAN VXCVI
CR1

Floating Convert From Integer
Doubleword X-form

fcfid FRT,FRB (Rc=0)
fcid FRT,FRB (Rc=1)

The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by FPSCR RN and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

FPSCR FPRF is set to the class and sign of the result. FPSCR FR is set if the result is incremented when rounded. FPSCR FI is set if the result is inexact.

Special Registers Altered:
FPRF FR FI
FX XX
CR1

Programming Note
Converting a signed integer word to double-precision floating-point can be accomplished by loading the word from storage using Load Float Word Algebraic Indexed and then using fcid.
Floating Convert From Integer Doubleword Unsigned X-form

[Category: Floating-Point.Phased-In]
fcfidu FRT,FRB (Rc=0)
fcfidu. FRT,FRB (Rc=1)

The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by FPSCR{RN}, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

FPSCR{FPRF} is set to the class and sign of the result. FPSCR{FR} is set if the result is incremented when rounded. FPSCR{FI} is set if the result is inexact.

Special Registers Altered:

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>XX</td>
<td>CR1 (if Rc=1)</td>
</tr>
</tbody>
</table>

Converting an unsigned integer word to double-precision floating-point can be accomplished by loading the word from storage using Load Float Word and Zero Indexed and then using fcfidu.

Floating Convert From Integer Doubleword Single X-form

[Category: Floating-Point.Phased-In]
fcfids FRT,FRB (Rc=0)
fcfids. FRT,FRB (Rc=1)

The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by FPSCR{RN}, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

FPSCR{FPRF} is set to the class and sign of the result. FPSCR{FR} is set if the result is incremented when rounded. FPSCR{FI} is set if the result is inexact.

Special Registers Altered:

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>XX</td>
<td>CR1 (if Rc=1)</td>
</tr>
</tbody>
</table>

Converting a signed integer word to single-precision floating-point can be accomplished by loading the word from storage using Load Float Word Algebraic Indexed and then using fcfids.
Floating Convert From Integer
Doubleword Unsigned Single X-form

[Category: Floating-Point.Phased-In]
fccfidus FRT,FRB (Rc=0)
fccfidus. FRT,FRB (Rc=1)

The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by FPSCR_RN, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

FPSCR_FPRF is set to the class and sign of the result. FPSCR_Fr is set if the result is incremented when rounded. FPSCR_FI is set if the result is inexact.

Special Registers Altered:

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX</td>
<td>XX</td>
<td>CR1</td>
</tr>
</tbody>
</table>

Programming Note

Converting a unsigned integer word to single-precision floating-point can be accomplished by loading the word from storage using Load Float Word and Zero Indexed and then using fccfidus.

4.6.7.3 Floating Round to Integer Instructions

The Floating Round to Integer instructions provide direct support for rounding functions found in high level languages. For example, frin, friz, frip, and frim implement C++ round(), trunc(), ceil(), and floor(), respectively. Note that frin does not implement the IEEE Round to Nearest function, which is often further described as “ties to even.” The rounding performed by these instructions is described fully in Section A.4, “Floating-Point Round to Integer Model” on page 694.

Programming Note

These instructions set FPSCR_FR1 to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of FPSCR_XX. Furthermore, it is believed that most programs do not need to know whether these rounding operations produce inexact or rounded results. If it is necessary to determine whether the result is inexact or rounded, software must compare the result with the original source operand.
Floating Round to Integer Nearest X-form

frin  FRT,FRB  (Rc=0)
frin. FRT,FRB  (Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>FRB</th>
<th>392</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value as follows, with the result placed into register FRT. If the sign of the operand is positive, \((FRB) + 0.5\) is truncated to an integral value, otherwise \((FRB) - 0.5\) is truncated to an integral value.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{VE} = 1.

Special Registers Altered:
- FPRF FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)

Floating Round to Integer Plus X-form

frip  FRT,FRB  (Rc=0)
frip. FRT,FRB  (Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>FRB</th>
<th>456</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward +infinity, and the result is placed into register FRT.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{VE} = 1.

Special Registers Altered:
- FPRF FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)

Floating Round to Integer Toward Zero X-form

friz  FRT,FRB  (Rc=0)
friz. FRT,FRB  (Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>FRB</th>
<th>424</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward zero, and the result is placed into register FRT.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{VE} = 1.

Special Registers Altered:
- FPRF FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)

Floating Round to Integer Minus X-form

frim  FRT,FRB  (Rc=0)
frim. FRT,FRB  (Rc=1)

<table>
<thead>
<tr>
<th>63</th>
<th>FRT</th>
<th>FRB</th>
<th>488</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward -infinity, and the result is placed into register FRT.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when FPSCR_{VE} = 1.

Special Registers Altered:
- FPRF FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1 (if Rc = 1)
4.6.8 Floating-Point Compare Instructions

The floating-point Compare instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to −0). The comparison can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three to 0. The CR is set in the same way.

**Floating Compare Unordered X-form**

```
fcmpu BF,FRA,FRB
```

<table>
<thead>
<tr>
<th>CR field BF</th>
<th>63</th>
<th>BF // FRA</th>
<th>16</th>
<th>21</th>
<th>0</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (FRA) is a NaN or (FRB) is a NaN then c ← 0b0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (FRA) &lt; (FRB) then c ← 0b1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (FRA) &gt; (FRB) then c ← 0b0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>c ← 0b0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPCC ← c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR4×BF:4×BF+3 ← c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (FRA) is an SNaN or (FRB) is an SNaN then VXSNAN ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX
- VXSNAN

**Floating Compare Ordered X-form**

```
fcmpo BF,FRA,FRB
```

<table>
<thead>
<tr>
<th>CR field BF</th>
<th>63</th>
<th>BF // FRA</th>
<th>16</th>
<th>21</th>
<th>32</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (FRA) is a NaN or (FRB) is a NaN then c ← 0b0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (FRA) &lt; (FRB) then c ← 0b1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (FRA) &gt; (FRB) then c ← 0b0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>c ← 0b0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPCC ← c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR4×BF:4×BF+3 ← c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (FRA) is an SNaN or (FRB) is an SNaN then VXSNAN ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if VE = 0 then VXVC ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (FRA) is a QNaN or (FRB) is a QNaN then VXVC ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set and, if Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, then VXVC is set.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX
- VXSNAN
- VXVC
4.6.9 Floating-Point Select Instruction

Floating Select A-form

\[
\text{fsel} \quad \text{FRT}, \text{FRA}, \text{FRC}, \text{FRB} \quad \text{(Rc}=0) \\
\text{fsel.} \quad \text{FRT}, \text{FRA}, \text{FRC}, \text{FRB} \quad \text{(Rc}=1)
\]

\[
\begin{array}{cccccc}
0 & 63 & 6 & 11 & 16 & 21 & 23 & 26 & 31 \\
\text{FRT} & \text{FRA} & \text{FRB} & \text{FRC} & \text{Rc}
\end{array}
\]

if \((\text{FRA}) \geq 0.0\) then \(\text{FRT} \leftarrow (\text{FRC})\)  \\
else \(\text{FRT} \leftarrow (\text{FRB})\)

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is a NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e., regards \(+0\) as equal to \(-0\)).

Special Registers Altered:  
CR1  
(if Rc=1)

Programming Note

Examples of uses of this instruction can be found in Sections F.2, “Floating-Point Conversions [Category: Floating-Point]” on page 726 and F.3, “Floating-Point Selection [Category: Floating-Point]” on page 730.

Warning: Care must be taken in using \text{fsel} if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section F.3.4, “Notes” on page 730.
4.6.10 Floating-Point Status and Control Register Instructions

Every Floating-Point Status and Control Register instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a Floating-Point Status and Control Register instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the Floating-Point Status and Control Register instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the Floating-Point Status and Control Register instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the Floating-Point Status and Control Register instruction is initiated.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the Floating-Point Status and Control Register instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the Floating-Point Status and Control Register instruction has completed.

(Floating-point Storage Access instructions are not affected.)

The instruction descriptions in this section refer to "FPSCR fields," where FPSCR field k is FPSCR bits 4xk:4xk+3.

Move From FPSCR X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mffs FRT</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>mffs FRT</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>FRT</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Rc</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

The contents of the FPSCR are placed into register FRT.

Special Registers Altered:
- CR1 (if Rc=1)

Move to Condition Register from FPSCR X-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcrfs BF,BFA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>BF</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>BFA</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

The contents of FPSCR_{32:63} field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the FPSCR. If the FX bit is copied, it is set to 0 in the FPSCR.

Special Registers Altered:
- CR field BF
  - FX OX (if BFA=0)
  - UX ZX XX VXSNAN (if BFA=1)
  - VXISI VXIDI VXZDZ VXIMZ (if BFA=2)
  - VXVC (if BFA=3)
  - VXSOFT VXSQRT VXCVI (if BFA=5)
Move To FPSCR Field Immediate X-form

mtfsfi BF,U,W (Rc=0)
mtfsfi BF,U,W (Rc=1)

The value of the U field is placed into FPSCR field BF+8×(1-W).
FPSCR_FX is altered only if BF = 0 and W = 0.

Special Registers Altered:
FPSCR field BF + 8×(1-W)
CR1 (if Rc=1)

--- Programming Note ---

mtfsfi serves as both a basic and an extended mnemonic. The Assembler will recognize a mtfsfi mnemonic with three operands as the basic form, and a mtfsfi mnemonic with two operands as the extended form. In the extended form the W operand is omitted and assumed to be 0.

--- Programming Note ---

When FPSCR_{32:35} is specified, bits 32 (FX) and 35 (OX) are set to the values of U_3 and U_2 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_3 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 117, and not from U_{1:2}.

Move To FPSCR Fields XFL-form

mtfsf FLM,FRB,L,W (Rc=0)
mtfsf FLM,FRB,L,W (Rc=1)

The FPSCR is modified as specified by the FLM, L, and W fields.

L = 0
The contents of register FRB are placed into the FPSCR under control of the W field and the field mask specified by FLM. W and the field mask identify the 4-bit fields affected. Let i be an integer in the range 0-7. If FLM_i=1 then FPSCR field k is set to the contents of the corresponding field of register FRB, where k = i+8×(1-W).

L = 1
The contents of register FRB are placed into the FPSCR.
FPSCR_FX is not altered implicitly by this instruction.

Special Registers Altered:
FPSCR fields selected by mask, L, and W
CR1 (if Rc=1)

--- Programming Note ---

mtfsf serves as both a basic and an extended mnemonic. The Assembler will recognize a mtfsf mnemonic with four operands as the basic form, and a mtfsf mnemonic with two operands as the extended form. In the extended form the W and L operands are omitted and both are assumed to be 0.

--- Programming Note ---

Updating fewer than eight fields of the FPSCR may have substantially poorer performance on some implementations than updating eight fields or all of the fields.

--- Programming Note ---

If L=1 or if L=0 and FPSCR_{32:35} is specified, bits 32 (FX) and 35 (OX) are set to the values of (FRB)_{32} and (FRB)_{35} (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from (FRB)_{32} and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 117, and not from (FRB)_{33:34}.
### Move To FPSCR Bit 0 X-form

<table>
<thead>
<tr>
<th>mtsfb0</th>
<th>BT</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtsfb0.</td>
<td>BT</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>70</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Bit BT+32 of the FPSCR is set to 0.

**Special Registers Altered:**
- FPSCR bit BT+32
- CR1 (if Rc=1)

**Programming Note**
Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

### Move To FPSCR Bit 1 X-form

<table>
<thead>
<tr>
<th>mtsfb1</th>
<th>BT</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtsfb1.</td>
<td>BT</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>38</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Bit BT+32 of the FPSCR is set to 1.

**Special Registers Altered:**
- FPSCR bits BT+32 and FX
- CR1 (if Rc=1)

**Programming Note**
Bits 33 and 34 (FEX and VX) cannot be explicitly set.
Chapter 5. Vector Facility [Category: Vector]

5.1 Vector Facility Overview

This chapter describes the registers and instructions that make up the Vector Facility.

5.2 Chapter Conventions

5.2.1 Description of Instruction Operation

The following notation, in addition to that described in Section 1.3.2, is used in this chapter. Additional RTL functions are described in Appendix C.

- $x.bit[y]$  
  Return the contents of bit $y$ of $x$.

- $x.bit[y:z]$  
  Return the contents of bits $y:z$ of $x$.

- $x.byte[y]$  
  Return the contents of byte element $y$ of $x$.

- $x.byte[y:z]$  
  Return the contents of byte elements $y:z$ of $x$.

- $x.hword[y]$  
  Return the contents of halfword element $y$ of $x$.

- $x.hword[y:z]$  
  Return the contents of halfword elements $y:z$ of $x$.

- $x.word[y]$  
  Return the contents of word element $y$ of $x$.

- $x.word[y:z]$  
  Return the contents of word element $y:z$ of $x$.

- $x.dword[y:z]$  
  Return the contents of doubleword elements $y:z$ of $x$.

- $x ? y : z$  
  If the value of $x$ is true, then the value of $y$, otherwise the value $z$.

- $^+_int$  
  Integer addition.

- $^+_fp$  
  Floating-point addition.

- $^-fp$  
  Floating-point subtraction.

- $x_{sui}$  
  Multiplication of a signed-integer (first operand) by an unsigned-integer (second operand).

- $x_{fp}$  
  Floating-point multiplication.

- $=_{int}$  
  Integer equals relation.

- $=_{fp}$  
  Floating-point equals relation.

- $<_{ui}$, $\leq_{ui}$, $>_{ui}$, $\geq_{ui}$  
 Unsigned-integer comparison relations.

- $<_{si}$, $\leq_{si}$, $>_{si}$, $\geq_{si}$  
  Signed-integer comparison relations.

- $<_{fp}$, $\leq_{fp}$, $>_{fp}$, $\geq_{fp}$  
  Floating-point comparison relations.

- $LENGTH(x)$  
  Length of $x$, in bits. If $x$ is the word “element”, $LENGTH(x)$ is the length, in bits, of the element implied by the instruction mnemonic.
\( x \ll y \)
Result of shifting \( x \) left by \( y \) bits, filling vacated bits with zeros.
\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow (y < b) \ ? \ (x_{y:b-1} \lor 0) : b_0
\]

\( x \gg y \)
Result of shifting \( x \) right by \( y \) bits, filling vacated bits with zeros.
\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow (y < b) \ ? \ (0 \| x_{(b-y):b-1}) : b_0
\]

\( x \ggg y \)
Result of shifting \( x \) right by \( y \) bits, filling vacated bits with copies of bit 0 (sign bit) of \( x \).
\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow (y < b) \ ? \ (\bar{y}_0 \| x_{(b-y):b-1}) : b_0
\]

\( x \lll y \)
Result of rotating \( x \) left by \( y \) bits.
\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow x_{y:b-1} \lor x_{y:0}
\]

\( x \ggg y \)
Returns the contents of \( x \) rotated right by \( y \) bits.

\( \text{Chop}(x, y) \)
Result of extending the right-most \( y \) bits of \( x \) on the left with zeros.
\[
\text{result} \leftarrow x \& ((1<y)-1)
\]

\( \text{EXTZ}(x) \)
Result of extending \( x \) on the left with zeros.
\[
b \leftarrow \text{LENGTH}(x) \\
\text{result} \leftarrow x \& ((1<b)-1)
\]

\( \text{Clamp}(x, y, z) \)
x is interpreted as a signed integer. If the value of \( x \) is less than \( y \), then the value \( y \) is returned, else if the value of \( x \) is greater than \( z \), the value \( z \) is returned, else the value \( x \) is returned.
\[
\text{if} \ (x < y) \ \text{then} \\
\text{result} \leftarrow y \\
\text{VSCRSAT} \leftarrow 1 \\
\text{else if} \ (x > z) \ \text{then} \\
\text{result} \leftarrow z \\
\text{VSCRSAT} \leftarrow 1 \\
\text{else result} \leftarrow x
\]

\( \text{InvMixColumns}(x) \)
\[
do \ c = 0 \ to \ 3 \\
\text{result}.\text{word}[c].\text{byte}[0] = 0x0E \times \text{word}[c].\text{byte}[0] \ ^ \ 0x0B \times \text{word}[c].\text{byte}[1] \ ^ \ 0x0D \times \text{word}[c].\text{byte}[2] \ ^ \ 0x09 \times \text{word}[c].\text{byte}[3] \\
\text{result}.\text{word}[c].\text{byte}[1] = 0x09 \times \text{word}[c].\text{byte}[0] \ ^ \ 0x0E \times \text{word}[c].\text{byte}[1] \ ^ \ 0x0B \times \text{word}[c].\text{byte}[2] \ ^ \ 0x0D \times \text{word}[c].\text{byte}[3] \\
\text{result}.\text{word}[c].\text{byte}[2] = 0x0D \times \text{word}[c].\text{byte}[0] \ ^ \ 0x09 \times \text{word}[c].\text{byte}[1] \ ^ \ 0x0E \times \text{word}[c].\text{byte}[2] \ ^ \ 0x0B \times \text{word}[c].\text{byte}[3] \\
\text{result}.\text{word}[c].\text{byte}[3] = 0x0B \times \text{word}[c].\text{byte}[0] \ ^ \ 0x0D \times \text{word}[c].\text{byte}[1] \ ^ \ 0x09 \times \text{word}[c].\text{byte}[2] \ ^ \ 0x0E \times \text{word}[c].\text{byte}[3] \\
\text{end} \\
\text{return}(\text{result});
\]

where “\(^\times\)" is a GF(2^8) multiply, a binary polynomial multiplication reduced by modulo \(0x11b\).

The GF(2^8) multiply of \(0x09\times x\) can be expressed in minimized terms as the following.
\[
\text{product}.\text{bit}[0] = x.\text{bit}[0] \ ^ \ x.\text{bit}[3] \\
\text{product}.\text{bit}[1] = x.\text{bit}[1] \ ^ \ x.\text{bit}[4] \ ^ \ x.\text{bit}[0] \\
\text{product}.\text{bit}[2] = x.\text{bit}[2] \ ^ \ x.\text{bit}[5] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \\
\text{product}.\text{bit}[3] = x.\text{bit}[3] \ ^ \ x.\text{bit}[6] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \ ^ \ x.\text{bit}[2] \\
\text{product}.\text{bit}[4] = x.\text{bit}[4] \ ^ \ x.\text{bit}[7] \ ^ \ x.\text{bit}[0] \\
\text{product}.\text{bit}[5] = x.\text{bit}[5] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \\
\text{product}.\text{bit}[6] = x.\text{bit}[6] \ ^ \ x.\text{bit}[1] \ ^ \ x.\text{bit}[2] \\
\text{product}.\text{bit}[7] = x.\text{bit}[7] \ ^ \ x.\text{bit}[2]
\]

The GF(2^8) multiply of \(0x0b\times x\) can be expressed in minimized terms as the following.
\[
\text{product}.\text{bit}[0] = x.\text{bit}[0] \ ^ \ x.\text{bit}[3] \\
\text{product}.\text{bit}[1] = x.\text{bit}[1] \ ^ \ x.\text{bit}[4] \ ^ \ x.\text{bit}[0] \\
\text{product}.\text{bit}[2] = x.\text{bit}[2] \ ^ \ x.\text{bit}[3] \ ^ \ x.\text{bit}[5] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \\
\text{product}.\text{bit}[3] = x.\text{bit}[3] \ ^ \ x.\text{bit}[4] \ ^ \ x.\text{bit}[6] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \ ^ \ x.\text{bit}[2] \\
\text{product}.\text{bit}[4] = x.\text{bit}[4] \ ^ \ x.\text{bit}[5] \ ^ \ x.\text{bit}[7] \ ^ \ x.\text{bit}[2] \\
\text{product}.\text{bit}[5] = x.\text{bit}[5] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \ ^ \ x.\text{bit}[2] \\
\text{product}.\text{bit}[6] = x.\text{bit}[6] \ ^ \ x.\text{bit}[7] \ ^ \ x.\text{bit}[0] \ ^ \ x.\text{bit}[1] \ ^ \ x.\text{bit}[2] \\
\text{product}.\text{bit}[7] = x.\text{bit}[7] \ ^ \ x.\bit[0] \ ^ \ x.\text{bit}[2]
The GF(2^8) multiply of 0x0D•x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= x.\text{bit}[0] \oplus x.\text{bit}[2] \oplus x.\text{bit}[3] \\
\text{product.bit}[1] &= x.\text{bit}[1] \oplus x.\text{bit}[3] \oplus x.\text{bit}[4] \oplus x.\text{bit}[0] \\
\text{product.bit}[2] &= x.\text{bit}[2] \oplus x.\text{bit}[4] \oplus x.\text{bit}[5] \oplus x.\text{bit}[1] \\
\text{product.bit}[3] &= x.\text{bit}[3] \oplus x.\text{bit}[5] \oplus x.\text{bit}[6] \oplus x.\text{bit}[0] \oplus x.\text{bit}[2] \\
\text{product.bit}[4] &= x.\text{bit}[4] \oplus x.\text{bit}[6] \oplus x.\text{bit}[7] \oplus x.\text{bit}[0] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[5] &= x.\text{bit}[5] \oplus x.\text{bit}[7] \oplus x.\text{bit}[1] \\
\text{product.bit}[6] &= x.\text{bit}[6] \oplus x.\text{bit}[0] \oplus x.\text{bit}[2] \\
\text{product.bit}[7] &= x.\text{bit}[7] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2]
\end{align*}
\]

The GF(2^8) multiply of 0x0E•x can be expressed in minimized terms as the following.

\[
\begin{align*}
\text{product.bit}[0] &= x.\text{bit}[1] \oplus x.\text{bit}[2] \oplus x.\text{bit}[3] \\
\text{product.bit}[1] &= x.\text{bit}[2] \oplus x.\text{bit}[3] \oplus x.\text{bit}[4] \oplus x.\text{bit}[0] \\
\text{product.bit}[2] &= x.\text{bit}[3] \oplus x.\text{bit}[4] \oplus x.\text{bit}[5] \oplus x.\text{bit}[1] \\
\text{product.bit}[3] &= x.\text{bit}[4] \oplus x.\text{bit}[5] \oplus x.\text{bit}[6] \oplus x.\text{bit}[2] \\
\text{product.bit}[4] &= x.\text{bit}[5] \oplus x.\text{bit}[6] \oplus x.\text{bit}[7] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2] \\
\text{product.bit}[5] &= x.\text{bit}[6] \oplus x.\text{bit}[7] \oplus x.\text{bit}[1] \\
\text{product.bit}[6] &= x.\text{bit}[7] \oplus x.\text{bit}[2] \\
\text{product.bit}[7] &= x.\text{bit}[0] \oplus x.\text{bit}[1] \oplus x.\text{bit}[2]
\end{align*}
\]

InvShiftRows(x)

\[
\begin{align*}
\text{result.word}[0].\text{byte}[0] &= x.\text{word}[0].\text{byte}[0] \\
\text{result.word}[1].\text{byte}[0] &= x.\text{word}[1].\text{byte}[0] \\
\text{result.word}[2].\text{byte}[0] &= x.\text{word}[2].\text{byte}[0] \\
\text{result.word}[3].\text{byte}[0] &= x.\text{word}[3].\text{byte}[0] \\
\text{result.word}[0].\text{byte}[1] &= x.\text{word}[3].\text{byte}[1] \\
\text{result.word}[1].\text{byte}[1] &= x.\text{word}[0].\text{byte}[1] \\
\text{result.word}[2].\text{byte}[1] &= x.\text{word}[1].\text{byte}[1] \\
\text{result.word}[3].\text{byte}[1] &= x.\text{word}[2].\text{byte}[1] \\
\text{result.word}[0].\text{byte}[2] &= x.\text{word}[2].\text{byte}[2] \\
\text{result.word}[1].\text{byte}[2] &= x.\text{word}[3].\text{byte}[2] \\
\text{result.word}[2].\text{byte}[2] &= x.\text{word}[0].\text{byte}[2] \\
\text{result.word}[3].\text{byte}[2] &= x.\text{word}[1].\text{byte}[2] \\
\text{result.word}[0].\text{byte}[3] &= x.\text{word}[1].\text{byte}[3] \\
\text{result.word}[1].\text{byte}[3] &= x.\text{word}[2].\text{byte}[3] \\
\text{result.word}[2].\text{byte}[3] &= x.\text{word}[3].\text{byte}[3] \\
\text{result.word}[3].\text{byte}[3] &= x.\text{word}[0].\text{byte}[3]
\end{align*}
\]

return(result)
InvSubBytes(x)

InvSBOX.byte[256] = { 0x52,0x09,0x6A,0x95,0x02,0知らない,0x33,0x99,0x6F,0x3E,0 이것은,0x4B,0x63,0x89,0x68,0x80,0x08,0x8B,0x37,0x7E,0x35,0x41,0x98,0x05,0x0C,0x5C,0x60,0x51,여기,0x7F,0x64,0x55,0x76,0x0A,0x75,0x3C,0x06,0x3F,0x54,0x1F,0x62,0x0E,0x03,0x0F,0x2A,0x71,0x70,0x01,0x73,0x73,0x03,0x26,0x1E,0x08,0x5B,0x65,여기,0x2C,0x15,0x0D,0x67,0x23,0x02,0x72,여기,0x20,0x01,0x0F,0x00,0x6F,0x02,여기,0x1F,0x51,여기,0x12,여기,0x10,여기,0x0D,여기,0x11,여기,0x16,여기,0x0E,여기,0x14,여기,0x13,여기,0x17,여기,0x18,여기,0x15,여기,0x19,여기,0x0C,여기,0x0B,여기,0x0A,여기,0x09,여기,0x07,여기,0x05,여기,0x04,여기,0x03,여기,0x02,여기,0x01,여기,0x00,여기,0x21,여기,0x22,여기,0x23,여기,0x24,여기,0x25,여기,0x26,여기,0x27,여기,0x28,여기,0x29,여기,0x2A,여기,0x2B,여기,0x2C,여기,0x2D,여기,0x2E,여기,0x2F,여기,0x30,여기,0x31,여기,0x32,여기,0x33,여기,0x34,여기,0x35,여기,0x36,여기,0x37,여기,0x38,여기,0x39,여기,0x3A,여기,0x3B,여기,0x3C,여기,0x3D,여기,0x3E,여기,0x3F }

do i = 0 to 15
  result.byte[i] = InvSBOX.byte[x.byte[i]]
end
return(result)

MixColumns(x)

do c = 0 to 3
  result.word[c].byte[0] = 0x02*x.word[c].byte[0] ^ 0x03*x.word[c].byte[1] ^ x.word[c].byte[2] ^ x.word[c].byte[3]
  result.word[c].byte[1] = x.word[c].byte[0] ^ 0x02*x.word[c].byte[1] ^ 0x03*x.word[c].byte[2] ^ x.word[c].byte[3]
end
return(result)

The GF(2^8) multiply of 0x02•x can be expressed in minimized terms as the following.

product.bit[0] = x.bit[1]
product.bit[7] = x.bit[0]

The GF(2^8) multiply of 0x03•x can be expressed in minimized terms as the following.

product.bit[0] = x.bit[0] ^ x.bit[1]
product.bit[7] = x.bit[0]
ShiftRows(x)

result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[1].byte[1]
result.word[3].byte[1] = x.word[0].byte[1]


result.word[1].byte[3] = x.word[0].byte[3]

return(result)

Signed_BCD_Add(x,y,z)

Let x and y be 31-digit signed decimal values.

Performs a signed decimal addition of x and y.

If the unbounded result is equal to zero, eq_flag is set to 1. Otherwise, eq_flag is set to 0.
If the unbounded result is greater than zero, gt_flag is set to 1. Otherwise, gt_flag is set to 0.
If the unbounded result is less than zero, lt_flag is set to 1. Otherwise, lt_flag is set to 0.

If the magnitude of the unbounded result is greater than \(10^{31}-1\), ox_flag is set to 1. Otherwise, ox_flag is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if z=0.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if z=1.
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatenated with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and inv_flag is set to 1 and lt_flag, gt_flag and eq_flag are set to 0. Otherwise, inv_flag is set to 0.
Signed_BCD_Subtract(x,y,z)

Let \( x \) and \( y \) be 31-digit signed decimal values.

Performs a signed decimal subtract of \( y \) from \( x \).

If the unbounded result is equal to zero, \( eq \_flag \) is set to 1. Otherwise, \( eq \_flag \) is set to 0.
If the unbounded result is greater than zero, \( gt \_flag \) is set to 1. Otherwise, \( gt \_flag \) is set to 0.
If the unbounded result is less than zero, \( lt \_flag \) is set to 1. Otherwise, \( lt \_flag \) is set to 0.

If the magnitude of the unbounded result is greater than \( 10^{31} \), \( ox \_flag \) is set to 1. Otherwise, \( ox \_flag \) is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if \( z=0 \).
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if \( z=1 \).
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatenated with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and \( inv \_flag \) is set to 0. Otherwise, \( inv \_flag \) is set to 0.

SubBytes(x)

\[
\text{do i = 0 to 15}
\]
\[
\text{result.byte[i] = SBOX.byte[x.byte[i]]}
\]
\[
\text{end}
\]
\[
\text{return(result)}
\]

RoundToSPIntCeil(x)

The value \( x \) if \( x \) is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than \( x \).

RoundToSPIntFloor(x)

The value \( x \) if \( x \) is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than \( x \).

RoundToSPIntNear(x)

The value \( x \) if \( x \) is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to \( x \) (in case of a tie, the even single-precision floating-point integer is used).

RoundToSPIntTrunc(x)

The value \( x \) if \( x \) is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than \( x \) if \( x>0 \), or the smallest single-precision floating-point integer that is greater than \( x \) if \( x<0 \).

RoundToNearSP(x)

The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result \( x \) (in case of a tie, the single-precision floating-point value with the least-significant bit equal to 0 is used).
ReciprocalEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the single-precision floating-point number x.

ReciprocalSquareRootEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the square root of the single-precision floating-point number x.

LogBase2EstimateSP(x)
A single-precision floating-point estimate of the base 2 logarithm of the single-precision floating-point number x.

Power2EstimateSP(x)
A single-precision floating-point estimate of the 2 raised to the power of the single-precision floating-point number x.

Figure 63. Vector Register elements

| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 127 |
5.3 Vector Facility Registers

5.3.1 Vector Registers

There are 32 Vector Registers (VRs), each containing 128 bits. See Figure 64. All computations and other data manipulation are performed on data residing in Vector Registers, and results are placed into a VR.

![Figure 64. Vector Registers](image)

Depending on the instruction, the contents of a Vector Register are interpreted as a sequence of equal-length elements (bytes, halfwords, or words) or as a quadword. Each of the elements is aligned at its natural boundary within the Vector Register, as shown in Figure 63. Many instructions perform a given operation in parallel on all elements in a Vector Register. Depending on the instruction, a byte, halfword, or word element can be interpreted as a signed-integer, an unsigned-integer, or a logical value; a word element can also be interpreted as a single-precision floating-point value. In the instruction descriptions, phrases like "signed-integer word element" are used as shorthand for "word element, interpreted as a signed-integer".

Load and Store instructions are provided that transfer a byte, halfword, word, or quadword between storage and a Vector Register.

5.3.2 Vector Status and Control Register

The Vector Status and Control Register (VSCR) is a special 32-bit register (not an SPR) that is read and written in a manner similar to the FPSCR in the Power ISA scalar floating-point unit. Special instructions (mvflscr and mtvflscr) are provided to move the VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right justified in the 128-bit vector register. When moved to a vector register, bits 0:95 of the vector register are cleared (set to 0).

![Figure 65. Vector Status and Control Register](image)

The bit definitions for the VSCR are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>96:110</td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td>Vector Non-Java Mode (NJ)</td>
</tr>
<tr>
<td></td>
<td>This bit controls how denormalized values are</td>
</tr>
<tr>
<td></td>
<td>handled by Vector Floating-Point instructions.</td>
</tr>
<tr>
<td></td>
<td>0 Denormalized values are handled as specified</td>
</tr>
<tr>
<td></td>
<td>by Java and the IEEE standard; see Section 5.6.1.</td>
</tr>
<tr>
<td></td>
<td>1 If an element in a source VR contains a</td>
</tr>
<tr>
<td></td>
<td>denormalized value, the value 0 is used instead.</td>
</tr>
<tr>
<td></td>
<td>If an instruction causes an Underflow Exception,</td>
</tr>
<tr>
<td></td>
<td>the corresponding element in the target VR is</td>
</tr>
<tr>
<td></td>
<td>set to 0. In both cases the 0 has the same sign</td>
</tr>
<tr>
<td></td>
<td>as the denormalized or underflowing value.</td>
</tr>
<tr>
<td>112:126</td>
<td>Reserved</td>
</tr>
<tr>
<td>127</td>
<td>Vector Saturation (SAT)</td>
</tr>
<tr>
<td></td>
<td>Every vector instruction having &quot;Saturate&quot; in its</td>
</tr>
<tr>
<td></td>
<td>name implicitly sets this bit to 1 if any result</td>
</tr>
<tr>
<td></td>
<td>of that instruction &quot;saturates&quot;; see Section 5.8.</td>
</tr>
<tr>
<td></td>
<td>mtvflscr can alter this bit explicitly. This bit</td>
</tr>
<tr>
<td></td>
<td>is sticky; that is, once set to 1 it remains</td>
</tr>
<tr>
<td></td>
<td>set to 1 until it is set to 0 by an mtvflscr</td>
</tr>
<tr>
<td></td>
<td>instruction.</td>
</tr>
</tbody>
</table>
After the `mfvscr` instruction executes, the result in the target vector register will be architecturally precise. That is, it will reflect all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow, and further, it will not reflect any SAT updates that may be made to it by vector instructions logically following it in the program flow. To implement this, processors may choose to make the `mfvscr` instruction execute serializing within the vector unit, meaning that it will stall vector instruction execution until all preceding vector instructions are complete and have updated the architectural machine state. This is permitted in order to simplify implementation of the sticky status bit (SAT) which would otherwise be difficult to implement in an out-of-order execution machine. The implication of this is that reading the VSCR can be much slower than typical Vector instructions, and therefore care must be taken in reading it, as advised in Section 5.5.1, to avoid performance problems.

The `mtvscr` is context synchronizing. This implies that all Vector instructions logically preceding an `mtvscr` in the program flow will execute in the architectural context (NJ mode) that existed prior to completion of the `mtvscr`, and that all instructions logically following the `mtvscr` will execute in the new context (NJ mode) established by the `mtvscr`.

### 5.3.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register in the fixed-point processor provided for application and operating system use; see Section 3.2.3.

---

**Programming Note**

The VRSAVE register can be used to indicate which VRs are currently being used by a program. If this is done, the operating system could save only those VRs when an “interrupt” occurs (see Book III), and could restore only those VRs when resuming the interrupted program.

If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VR is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent.

Some operating systems save and restore VRSAVE only for programs that also use other vector registers.

### 5.4 Vector Storage Access Operations

The Vector Storage Access instructions provide the means by which data can be copied from storage to a Vector Register or from a Vector Register to storage. Instructions are provided that access byte, halfword, word, and quadword storage operands. These instructions differ from the fixed-point and floating-point Storage Access instructions in that vector storage operands are assumed to be aligned, and vector storage accesses are performed as if the appropriate number of low-order bits of the specified effective address (EA) were zero. For example, the low-order bit of EA is ignored for halfword Vector Storage Access instructions, and the low-order four bits of EA are ignored for quadword Vector Storage Access instructions. The effect is to load or store the storage operand of the specified length that contains the byte addressed by EA.

If a storage operand is unaligned, additional instructions must be used to ensure that the operand is correctly placed in a Vector Register or in storage. Instructions are provided that shift and merge the contents of two Vector Registers, such that an unaligned quadword storage operand can be copied between storage and the Vector Registers in a relatively efficient manner.

As shown in Figure 63, the elements in Vector Registers are numbered; the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15. The numbering affects the values that must be placed into the permute control vector for the Vector Permute instruction in order for that instruction to achieve the desired effects, as illustrated by the examples in the following subsections.

A vector quadword Load instruction for which the effective address (EA) is quadword-aligned places the byte in storage addressed by EA into byte element 0 of the target Vector Register, the byte in storage addressed by EA+1 into byte element 1 of the target Vector Register, etc. Similarly, a vector quadword Store instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source Vector Register into the byte in storage addressed by EA, the contents of byte element 1 of the source Vector Register into the byte in storage addressed by EA+1, etc.
Figure 66 shows an aligned quadword in storage. Figure 67 shows the result of loading that quadword into a Vector Register, or, equivalently, shows the contents that must be in a Vector Register if storing that Vector Register is to produce the storage contents shown in Figure 66.

When an aligned byte, halfword, or word storage operand is loaded into a Vector Register, the element (byte, halfword, or word respectively) that receives the data is the element that would have received the data had the entire aligned quadword containing the storage operand addressed by EA been loaded. Similarly, when a byte, halfword, or word element in a Vector Register is stored into an aligned storage operand (byte, halfword, or word respectively), the element selected to be stored is the element that would have been stored into the storage operand addressed by EA had the entire Vector Register been stored to the aligned quadword containing the storage operand addressed by EA. (Byte storage operands are always aligned.)

For aligned byte, halfword, and word storage operands, if the corresponding element number is known when the program is written, the appropriate Vector Splat and Vector Permute instructions can be used to copy or replicate the data contained in the storage operand after loading the operand into a Vector Register. An example of this is given in the Programming Note for Vector Splat; see page 196. Another example is to replicate the element across an entire Vector Register before storing it into an arbitrary aligned storage operand of the same length; the replication ensures that the correct data are stored regardless of the offset of the storage operand in its aligned quadword in storage.
5.4.1 Accessing Unaligned Storage Operands

Figure 68 shows an unaligned quadword storage operand that spans two aligned quadwords. In the remainder of this section, the aligned quadword that contains the most significant bytes of the unaligned quadword is called the most significant quadword (MSQ) and the aligned quadword that contains the least significant bytes of the unaligned quadword is called the least significant quadword (LSQ). Because the Vector Storage Access instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a Vector Register using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned halfword and word storage operands.

Programming Note

The sequence of instructions given below is one approach that can be used to load the unaligned quadword shown in Figure 68 into a Vector Register. In Figure 69 Vhi and Vlo are the Vector Registers that will receive the most significant quadword and least significant quadword respectively. VRT is the target Vector Register.

After the two quadwords have been loaded into Vhi and Vlo, using Load Vector Indexed instructions, the alignment is performed by shifting the 32-byte quantity Vhi || Vlo left by an amount determined by the address of the first byte of the desired data. The shifting is done using a Vector Permute instruction for which the permute control vector is generated by a Load Vector for Shift Left instruction. The Load Vector for Shift Left instruction uses the same address specification as the Load Vector Indexed instruction that loads the Vhi register; this is the address of the desired unaligned quadword.

The following sequence of instructions copies the unaligned quadword storage operand into register Vt.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb      # load MSQ
lvx    Vlo,0,Rb      # load LSQ
lvsl   Vp,0,Rb       # set permute control vector
addi   Rb,Rb,16      # address of LSQ
lvx    Vlo,0,Rb      # load LSQ
vperm  Vt,Vhi,Vlo,Vp # align the data
```

The procedure for storing an unaligned quadword is essentially the reverse of the procedure for loading one. However, a read-modify-write sequence is required that inserts the source quadword into two aligned quadwords in storage. The quadword to be stored is assumed to be in Vs; see Figure 69. The contents of Vs are shifted right and split into two parts, each of which is merged (using a Vector Select instruction) with the current contents of the two aligned quadwords (MSQ and LSQ) that will contain the most significant bytes and least significant bytes, respectively, of the unaligned quadword. The resulting two quadwords are stored using Store Vector Indexed instructions. A Load Vector for Shift Right instruction is used to generate the permute control vector that is used for the shifting. A single register is used for the “shifted” contents; this is possible because the “shifting” is done by means of a right rotation. The rotation is accomplished by specifying Vs for both components of the Vector Permute instruction. In addition, the same permute control vector is used on a sequence of 1s and 0s to generate the mask used by the Vector Select instructions that do the merging.

The following sequence of instructions copies the contents of Vs into an unaligned quadword in storage.

```
# Assumptions:
# Rb := 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb      # load current MSQ
lvx    Vlo,0,Rb      # load current LSQ
lvsr   Vp,0,Rb       # set permute control vector
addi   Rb,Rb,16      # address of LSQ
vperm  Vmask,V0s,V1s,Vp # generate the select mask
vperm  Vs,Vs,Vs,Vp    # right rotate the data
vsel   Vlo,Vs,Vlo,Vmask # insert LSQ component
vsel   Vhi,Vhi,Vso,Vmask # insert MSQ component
stvx   Vlo,0,Rb      # store LSQ
addi   Rb,Rb,-16     # address of MSQ
stvx   Vhi,0,Rb      # store MSQ
```
5.5 Vector Integer Operations

Many of the instructions that produce fixed-point integer results have the potential to compute a result value that cannot be represented in the target format. When this occurs, this unrepresentable intermediate value is converted to a representable result value using one of the following methods.

1. The high-order bits of the intermediate result that do not fit in the target format are discarded. This method is used by instructions having names that include the word "Modulo".

2. The intermediate result is converted to the nearest value that is representable in the target format (i.e., to the minimum or maximum representable value, as appropriate). This method is used by instructions having names that include the word "Saturate". An intermediate result that is forced to the minimum or maximum representable value as just described is said to "saturate".

An instruction for which an intermediate result saturates causes VSCR\textsubscript{SAT} to be set to 1; see Section 5.3.2.

3. If the intermediate result includes non-zero fraction bits it is rounded up to the nearest fixed-point integer value. This method is used by the six Vector Average Integer instructions and by the Vector Multiply-High-Round-Add Signed Halfword Saturate instruction. The latter instruction then uses method 2, if necessary.

Programming Note

Because VSCR\textsubscript{SAT} is sticky, it can be used to detect whether any instruction in a sequence of “Saturate”-type instructions produced an inexact result due to saturation. For example, the contents of the VSCR can be copied to a VR (mfvscr), bits other than the SAT bit can be cleared in the VR (vand with a constant), the result can be compared to zero setting CR6 (vcmpequb), and a branch can be taken according to whether VSCR\textsubscript{SAT} was set to 1 (Branch Conditional that tests CR field 6).

Testing VSCR\textsubscript{SAT} after each “Saturate”-type instruction would degrade performance considerably. Alternative techniques include the following:

- Retain sufficient information at "checkpoints" that the sequence of computations performed between one checkpoint and the next can be redone (more slowly) in a manner that detects exactly when saturation occurs. Test VSCR\textsubscript{SAT} only at checkpoints, or when redoing a sequence of computations that saturated.

- Perform intermediate computations using an element length sufficient to prevent saturation, and then use a Vector Pack Integer Saturate instruction to pack the final result to the desired length. (Vector Pack Integer Saturate causes results to saturate if necessary, and sets VSCR\textsubscript{SAT} to 1 if any result saturates.)

5.5.1 Integer Saturation

Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero (0) on underflow and to the maximum positive integer value ($2^{n-1}$, e.g. 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number ($-2^{n-1}$, e.g. -128 for byte fields) on underflow, and to the largest representable positive number ($2^{n-1}$-1, e.g. +127 for byte fields) on overflow.
In most cases, the simple maximum/minimum saturation performed by the vector instructions is adequate. However, sometimes, e.g. in the creation of very high quality images, more complex saturation functions must be applied. To support this, the Vector facility provides a mechanism for detecting that saturation has occurred. The VSCR has a bit, the SAT bit, which is set to a one (1) anytime any field in a saturating instruction saturates. The SAT bit can only be cleared by explicitly writing zero to it. Thus SAT accumulates a summary result of any integer overflow or underflow that occurs on a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned 0+0=0 and unsigned byte 1+254=255, are not considered saturation conditions and do not cause SAT to be set.

The SAT bit can be set by the following types of instructions:

- Move To VSCR
- Vector Add Integer with Saturation
- Vector Subtract Integer with Saturation
- Vector Multiply-Add Integer with Saturation
- Vector Multiply-Sum with Saturation
- Vector Sum-Across with Saturation
- Vector Pack with Saturation
- Vector Convert to Fixed-point with Saturation

Note that only instructions that explicitly call for “saturation” can set SAT. “Modulo” integer instructions and floating-point arithmetic instructions never set SAT.

---

**Programming Note**

The SAT state can be tested and used to alter program flow by moving the VSCR to a vector register (with `mfvscr`), then masking out bits 0:126 (to clear undefined and reserved bits) and performing a vector compare equal-to unsigned byte w/record (`vcmpequb`) with zero to get a testable value into the condition register for consumption by a subsequent branch.

Since `mfvscr` will be slow compared to other Vector instructions, reading and testing SAT after each instruction would be prohibitively expensive. Therefore, software is advised to employ strategies that minimize checking SAT. For example: checking SAT periodically and backtracking to the last checkpoint to identify exactly which field in which instruction saturated; or, working in an element size sufficient to prevent any overflow or underflow during intermediate calculations, then packing down to the desired element size as the final operation (the vector pack instruction saturates the results and updates SAT when a loss of significance is detected).
5.6 Vector Floating-Point Operations

5.6.1 Floating-Point Overview

Unless VSCR_{NJ}=1 (see Section 5.3.2), the floating-point model provided by the Vector Facility conforms to The Java Language Specification (hereafter referred to as "Java"), which is a subset of the default environment specified by the IEEE standard (i.e., by ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic"). For aspects of floating-point behavior that are not defined by Java but are defined by the IEEE standard, vector floating-point conforms to the IEEE standard. For aspects of floating-point behavior that are defined neither by Java nor by the IEEE standard but are defined by the "C9X Floating-Point Proposal" (hereafter referred to as "C9X"), vector floating-point conforms to C9X.

The single-precision floating-point data format, value representations, and computational models defined in Chapter 4. "Floating-Point Facility [Category: Floating-Point]" on page 115 apply to vector floating-point except as follows.

- In general, no status bits are set to reflect the results of floating-point operations. The only exception is that VSCR_{SAT} may be set by the Vector Convert To Fixed-Point Word instructions.

- With the exception of the two Vector Convert To Fixed-Point Word instructions and three of the four Vector Round To Floating-Point Integer instructions, all vector floating-point instructions that round use the rounding mode Round to Nearest.

- Floating-point exceptions (see Section 5.6.2) cannot cause the system error handler to be invoked.

Programming Note

If a function is required that is specified by the IEEE standard, is not supported by the Vector Facility, and cannot be emulated satisfactorily using the functions that are supported by the Vector Facility, the functions provided by the Floating-Point Facility should be used; see Chapter 4.

5.6.2 Floating-Point Exceptions

The following floating-point exceptions may occur during execution of vector floating-point instructions.

- NaN Operand Exception
- Invalid Operation Exception
- Zero Divide Exception
- Log of Zero Exception
- Overflow Exception
- Underflow Exception

If an exception occurs, a result is placed into the corresponding target element as described in the following subsections. This result is the default result specified by Java, the IEEE standard, or C9X, as applicable.

Recall that denormalized source values are treated as if they were zero when VSCR_{NJ}=1. This has the following consequences regarding exceptions.

- Exceptions that can be caused by a zero source value can be caused by a denormalized source value when VSCR_{NJ}=1.

- Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when VSCR_{NJ}=1.

5.6.2.1 NaN Operand Exception

A NaN Operand Exception occurs when a source value for any of the following instructions is a NaN.

- A vector instruction that would normally produce floating-point results
- Either of the two Vector Convert To Fixed-Point Word instructions
- Any of the four Vector Floating-Point Compare instructions

The following actions are taken:

If the vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is a Signaling NaN it is converted to the corresponding Quiet NaN (by setting the high-order bit of the fraction field to 1) before being placed into the target element.

if the element in VRA is a NaN then the result is that NaN
else if the element in VRB is a NaN then the result is that NaN
else if the element in VRC is a NaN

if a function is required that is specified by the IEEE standard, is not supported by the Vector Facility, and cannot be emulated satisfactorily using the functions that are supported by the Vector Facility, the functions provided by the Floating-Point Facility should be used; see Chapter 4.
then the result is that NaN
else if Invalid Operation exception
    (Section 5.6.2.2)
then the result is the QNaN 0x7FC0_0000

If the instruction is either of the two Vector Convert To Fixed-Point Word instructions, the corresponding result is 0x0000_0000. VSCR\textsubscript{SAT} is not affected.

If the instruction is Vector Compare Bounds Floating-Point, the corresponding result is 0xC000_0000.

If the instruction is one of the other Vector Floating-Point Compare instructions, the corresponding result is 0x0000_0000.

### 5.6.2.2 Invalid Operation Exception

An Invalid Operation Exception occurs when a source value or set of source values is invalid for the specified operation. The invalid operations are:

- Magnitude subtraction of infinities
- Multiplication of infinity by zero
- Reciprocal square root estimate of a negative, nonzero number or -infinity.
- Log base 2 estimate of a negative, nonzero number or -infinity.

The corresponding result is the QNaN 0x7FC0_0000.

### 5.6.2.3 Zero Divide Exception

A Zero Divide Exception occurs when a Vector Reciprocal Estimate Floating-Point or Vector Reciprocal Square Root Estimate Floating-Point instruction is executed with a source value of zero.

The corresponding result is an infinity, where the sign is the sign of the source value.

### 5.6.2.4 Log of Zero Exception

A Log of Zero Exception occurs when a Vector Log Base 2 Estimate Floating-Point instruction is executed with a source value of zero.

The corresponding result is -Infinity.

### 5.6.2.5 Overflow Exception

An Overflow Exception occurs under either of the following conditions.

- For a vector instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite floating-point number for the target floating-point format.
- For either of the two Vector Convert To Fixed-Point Word instructions, either a source value is an infinity or the product of a source value and \(2^{\text{UM}}\) is a number too large in magnitude to be represented in the target fixed-point format.

The following actions are taken:

1. If the vector instruction would normally produce floating-point results, the corresponding result is an infinity, where the sign is the sign of the intermediate result.
2. If the instruction is Vector Convert To Unsigned Fixed-Point Word Saturate, the corresponding result is 0xFFFF_FFFF if the source value is a positive number or +infinity, and is 0x0000_0000 if the source value is a negative number or -infinity. VSCR\textsubscript{SAT} is set to 1.
3. If the instruction is Vector Convert To Signed Fixed-Point Word Saturate, the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +infinity, and is 0x8000_0000 if the source value is a negative number or -infinity. VSCR\textsubscript{SAT} is set to 1.

### 5.6.2.6 Underflow Exception

An Underflow Exception can occur only for vector instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded is less in magnitude than the smallest normalized floating-point number for the target floating-point format.

The following actions are taken:

1. If VSCR\textsubscript{NJ}=0, the corresponding result is the value produced by denormalizing and rounding the intermediate result.
2. If VSCR\textsubscript{NJ}=1, the corresponding result is a zero, where the sign is the sign of the intermediate result.
5.7 Vector Storage Access Instructions

The Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 26. The low-order bits of the EA that would correspond to an unaligned storage operand are ignored.

The Load Vector Element Indexed and Store Vector Element Indexed instructions transfer a byte, halfword, or word element between storage and a Vector Register. The Load Vector Indexed and Store Vector Indexed instructions transfer an aligned quadword between storage and a Vector Register.

5.7.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.
5.7.2 Vector Load Instructions

The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into register VRT.

**Load Vector Element Byte Indexed X-form**

\[ \text{lvebx VRT,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>7</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
eb ← EA60:63
VRT ← undefined
if Big-Endian byte ordering then
VRT8×eb:8×eb+7 ← MEM(EA,1)
else
VRT120-(8×eb):127-(8×eb) ← MEM(EA,1)

Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte eb of register VRT. The remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT. The remaining bytes in register VRT are set to undefined values.

**Special Registers Altered:**

None

**Load Vector Element Halfword Indexed X-form**

\[ \text{lvehx VRT,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>VRT</th>
<th>RA</th>
<th>RB</th>
<th>39</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA60:63
VRT ← undefined
if Big-Endian byte ordering then
VRT8×eb:8×eb+15 ← MEM(EA,2)
else
VRT112-(8×eb):127-(8×eb) ← MEM(EA,2)

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

**Special Registers Altered:**

None

---

**Programming Note**

The Load Vector Element instructions load the specified element into the same location in the target register as the location into which it would be loaded using the Load Vector instruction.
**Load Vector Element Word Indexed X-form**

\[
\text{lvewx} \quad \text{VRT,RA,RB}
\]

\[
\begin{array}{cccccc}
0 & 31 & 6 & 11 & 16 & 21 & 71 \\
\hline
0 & 0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

- If RA = 0 then \( b \leftarrow 0 \)
- Else \( b \leftarrow |\text{RA}| \)
- \( \text{EA} \leftarrow (b + \text{RB}) \& 0xFFFF_FFFF_FFFF_FFFC \)
- \( \text{eb} \leftarrow \text{EA}[63:60] \)
- \( \text{VRT} \leftarrow \text{undefined} \)
- If Big-Endian byte ordering then
  - \( \text{VRT}_{8\times\text{eb}}:8\times\text{eb}+31 \leftarrow \text{MEM}(\text{EA}, 4) \)
- Else
  - \( \text{VRT}_{8\times\text{eb}}:8\times\text{eb}+31 \leftarrow \text{MEM}(\text{EA}, 4) \)

Let the effective address (EA) be the result of ANDing \( 0xFFFF_FFFF_FFFF_FFFC \) with the sum \((\text{RA}(0)+(\text{RB}))\).

Let \( \text{eb} \) be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte \( \text{eb} \) of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte \( \text{eb}+1 \) of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte \( \text{eb}+2 \) of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte \( \text{eb}+3 \) of register VRT,
- the remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte \( \text{eb} \) of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte \( \text{eb}+1 \) of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte \( \text{eb}+2 \) of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte \( \text{eb}+3 \) of register VRT,
- the remaining bytes in register VRT are set to undefined values.

**Special Registers Altered:**

None

**Load Vector Indexed X-form**

\[
\text{lvx} \quad \text{VRT,RA,RB}
\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & 21 & 31 \\
\hline
0 & 0 & 1 & 2 & 3 & 4
\end{array}
\]

- If RA = 0 then \( b \leftarrow 0 \)
- Else \( b \leftarrow |\text{RA}| \)
- \( \text{EA} \leftarrow b + \text{RB} \)
- \( \text{VRT} \leftarrow \text{MEM}(\text{EA} \& 0xFFFF_FFFF_FFFF_FFF0, 16) \)

Let the effective address (EA) be the sum \((\text{RA}(0)+(\text{RB}))\). The quadword in storage addressed by the result of EA ANDed with \( 0xFFFF_FFFF_FFFF_FFF0 \) is loaded into VRT.

**Special Registers Altered:**

None

**Load Vector Indexed LRU X-form**

\[
\text{lvxl} \quad \text{VRT,RA,RB}
\]

\[
\begin{array}{cccccc}
31 & 6 & 11 & 16 & 21 & 31 \\
\hline
0 & 0 & 1 & 2 & 3 & 4
\end{array}
\]

- If RA = 0 then \( b \leftarrow 0 \)
- Else \( b \leftarrow |\text{RA}| \)
- \( \text{EA} \leftarrow b + \text{RB} \)
- \( \text{VRT} \leftarrow \text{MEM}(\text{EA} \& 0xFFFF_FFFF_FFFF_FFF0, 16) \)
- \( \text{mark}_{\text{as_not Likely_to_be_needed}_again(_anytime_soon)}(\text{EA}) \)

Let the effective address (EA) be the sum \((\text{RA}(0)+(\text{RB}))\). The quadword in storage addressed by the result of EA ANDed with \( 0xFFFF_FFFF_FFFF_FFF0 \) is loaded into VRT.

\( \text{lvxl} \) provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

**Special Registers Altered:**

None
On some implementations, the hint provided by the \texttt{lvxl} instruction and the corresponding hint provided by the \texttt{stvxl}, \texttt{lvepxl}, and \texttt{stvepxl} instructions are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.

---

**Programming Note**

On some implementations, the hint provided by the \texttt{lvxl} instruction and the corresponding hint provided by the \texttt{stvxl}, \texttt{lvepxl}, and \texttt{stvepxl} instructions are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.
5.7.3 Vector Store Instructions

Some portion or all of the contents of VRS are stored into the aligned byte, halfword, word, or quadword in storage addressed by EA.

--- Programming Note ---

The Store Vector Element instructions store the specified element into the same storage location as the location into which it would be stored using the Store Vector instruction.

--- Store Vector Element Byte Indexed X-form ---

\[ \text{stvebx} \quad \text{VRS},\text{RA},\text{RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>VRS</th>
<th>RA</th>
<th>RB</th>
<th>135</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)

EA ← b + (RB)

if Big-Endian byte ordering then

[math] MEM(EA,1) ← VRS_{8\times eb:8\times eb+7} \]

else

[math] MEM(EA,1) ← VRS_{120-(8\times eb):127-(8\times eb)} \]

Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of byte eb of register VRS are placed in the byte in storage at address EA.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access, the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA.

Special Registers Altered:
None

--- Programming Note ---

Unless bits 60:63 of the address are known to match the byte offset of the subject byte element in register VRS, software should use \text{Vector Splat} to splat the subject byte element before performing the store.

--- Store Vector Element Halfword Indexed X-form ---

\[ \text{stvehx} \quad \text{VRS},\text{RA},\text{RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>VRS</th>
<th>RA</th>
<th>RB</th>
<th>167</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)

EA ← b + (RA)

if Big-Endian byte ordering then

[math] MEM(EA,2) ← VRS_{8\times eb:8\times eb+15} \]

else

[math] MEM(EA,2) ← VRS_{120-(8\times eb):127-(8\times eb)} \]

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

– the contents of byte eb of register VRS are placed in the byte in storage at address EA, and
– the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

– the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA, and
– the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1.

Special Registers Altered:
None

--- Programming Note ---

Unless bits 60:62 of the address are known to match the halfword offset of the subject halfword element in register VRS software should use \text{Vector Splat} to splat the subject halfword element before performing the store.
**Store Vector Element Word Indexed X-form**

`stvewx VRS,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>RB</th>
<th>199</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)

EA ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC

`eb ← EA[6:31]`

If Big-Endian byte ordering then

- `MEN(EA, 4) ← VRS[8×ab:8×ab+31]`

else

- `MEN(EA, 4) ← VRS[36−(8×ab):127−(8×ab)]`

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

- If Big-Endian byte ordering is used for the storage access,
  - the contents of byte eb of register VRS are placed in the byte in storage at address EA,
  - the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1,
  - the contents of byte eb+2 of register VRS are placed in the byte in storage at address EA+2, and
  - the contents of byte eb+3 of register VRS are placed in the byte in storage at address EA+3.

- If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,
  - the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA,
  - the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1,
  - the contents of byte 13-eb of register VRS are placed in the byte in storage at address EA+2, and
  - the contents of byte 12-eb of register VRS are placed in the byte in storage at address EA+3.

**Special Registers Altered:**

- None

---

**Store Vector Indexed X-form**

`stvx VRS,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>RB</th>
<th>231</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)

- MEN(EA & 0xFFFF_FFFF_FFFF_FFO, 16) ← (VRS)

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

**Special Registers Altered:**

- None

---

**Store Vector Indexed LRU X-form**

`stvxl VRS,RA,RB`

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>RA</th>
<th>RB</th>
<th>487</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

If RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)

- MEN(EA & 0xFFFF_FFFF_FFFF_FFO, 16) ← (VRS)
- `mark_as_not likely_to_be_needed_again_anytime_soon(EA)`

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

**stvxl** provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

**Special Registers Altered:**

- None

---

**Programming Note**

See the Programming Note for the `lvxl` instruction on page 182.

---

Unless bits 60:61 of the address are known to match the word offset of the subject word element in register VRS, software should use Vector Splat to splat the subject word element before performing the store.
5.7.4 Vector Alignment Support Instructions

**Programming Note**

The **lvsl** and **lvsr** instructions can be used to create the permute control vector to be used by a subsequent **vperm** instruction (see page 198). Let X and Y be the contents of register VRA and VRB specified by the **vperm**. The control vector created by **lvsl** causes the **vperm** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes. The control vector created by **lvsr** causes the **vperm** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

### Load Vector for Shift Left Indexed X-form

**lvsl**  
VRT, RA, RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← [RA]
sh ← (b + (RB))_{60:63}
switch(sh)
case(0x0): VRT ← 0x000102030405060708090A0B0C0D0E0F
case(0x1): VRT ← 0x0102030405060708090A0B0C0D0E0F10
case(0x2): VRT ← 0x02030405060708090A0B0C0D0E0F1011
case(0x3): VRT ← 0x030405060708090A0B0C0D0E0F101112
case(0x4): VRT ← 0x0405060708090A0B0C0D0E0F10111213
case(0x5): VRT ← 0x05060708090A0B0C0D0E0F1011121314
case(0x6): VRT ← 0x060708090A0B0C0D0E0F101112131415
case(0x7): VRT ← 0x0708090A0B0C0D0E0F10111213141516
case(0x8): VRT ← 0x08090A0B0C0D0E0F1011121314151617
case(0x9): VRT ← 0x090A0B0C0D0E0F101112131415161718
case(0xA): VRT ← 0x0A0B0C0D0E0F10111213141516171819
case(0xB): VRT ← 0x0B0C0D0E0F101112131415161718191A
case(0xC): VRT ← 0x0C0D0E0F101112131415161718191A1B
case(0xD): VRT ← 0x0D0E0F101112131415161718191A1B1C
case(0xE): VRT ← 0x0E0F101112131415161718191A1B1C1D
case(0xF): VRT ← 0x0F101112131415161718191A1B1C1D1E

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F.

Bytes sh to sh+15 of X are placed into VRT.

**Special Registers Altered:**

None

### Load Vector for Shift Right Indexed X-form

**lvsr**  
VRT, RA, RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>38</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← [RA]
sh ← (b + (RB))_{60:63}
switch(sh)
case(0x0): VRT ← 0x101112131415161718191A1B1C1D1E1F
case(0x1): VRT ← 0x0F101112131415161718191A1B1C1D1E
case(0x2): VRT ← 0x0E0F101112131415161718191A1B1C1D
case(0x3): VRT ← 0x0D0E0F101112131415161718191A1B1C
case(0x4): VRT ← 0x0C0D0E0F101112131415161718191A1B
case(0x5): VRT ← 0x0B0C0D0E0F101112131415161718191A
case(0x6): VRT ← 0x0A0B0C0D0E0F101112131415161718191A
case(0x7): VRT ← 0x090A0B0C0D0E0F101112131415161718191A
case(0x8): VRT ← 0x08090A0B0C0D0E0F101112131415161718191A
case(0x9): VRT ← 0x0708090A0B0C0D0E0F101112131415161718191A
case(0xA): VRT ← 0x060708090A0B0C0D0E0F101112131415161718191A
case(0xB): VRT ← 0x05060708090A0B0C0D0E0F101112131415161718191A
case(0xC): VRT ← 0x0405060708090A0B0C0D0E0F101112131415161718191A
case(0xD): VRT ← 0x030405060708090A0B0C0D0E0F101112131415161718191A
case(0xE): VRT ← 0x02030405060708090A0B0C0D0E0F10

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F.

Bytes 16-sh to 31-sh of X are placed into VRT.

**Special Registers Altered:**

None

---

Programming Note

Examples of uses of **lvsl**, **lvsr**, and **vperm** to load and store unaligned data are given in Section 5.4.1.

These instructions can also be used to rotate or shift the contents of a Vector Register left (**lvsl**) or right (**lvsr**) by sh bytes. For rotating, the Vector Register to be rotated should be specified as both register VRA and VRB for **vperm**. For shifting left, VRB for **vperm** should be a register containing all zeros and VRA should contain the value to be shifted, and vice versa for shifting right.

---

Programming Note

Examples of uses of **lvsl**, **lvsr**, and **vperm** to load and store unaligned data are given in Section 5.4.1.

These instructions can also be used to rotate or shift the contents of a Vector Register left (**lvsl**) or right (**lvsr**) by sh bytes. For rotating, the Vector Register to be rotated should be specified as both register VRA and VRB for **vperm**. For shifting left, VRB for **vperm** should be a register containing all zeros and VRA should contain the value to be shifted, and vice versa for shifting right.
5.8 Vector Permute and Formatting Instructions

5.8.1 Vector Pack and Unpack Instructions

**Vector Pack Pixel VX-form**

```
vpkpx VRT, VRA, VRB
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>782</td>
</tr>
</tbody>
</table>

Let the source vector be the concatenation of the contents of `VR[VRA]` followed by the contents of `VR[VRB]`.

For each integer value `i` from 0 to 7, do the following.

Word element `i` in the source vector is packed to produce a 16-bit value as described below.

- bit 7 of the first byte (bit 7 of the word)
- bits 0:4 of the second byte (bits 8:12 of the word)
- bits 0:4 of the third byte (bits 16:20 of the word)
- bits 0:4 of the fourth byte (bits 24:28 of the word)

The result is placed into halfword element `i` of `VR[VRT].`

**Special Registers Altered:**

None

**Programming Note**

Each source word can be considered to be a 32-bit "pixel", consisting of four 8-bit "channels". Each target halfword can be considered to be a 16-bit pixel, consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.

**Vector Pack Signed Doubleword Signed Saturate VX-form**

```
vpksdss VRT, VRA, VRB
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>1486</td>
</tr>
</tbody>
</table>

Let doubleword elements 0 and 1 of `src` be the contents of `VR[VRA].`

Let doubleword elements 2 and 3 of `src` be the contents of `VR[VRB].`

For each integer value `i` from 0 to 3, do the following.

The signed integer value in doubleword element `i` of `src` is placed into word element `i` of `VR[VRT]` in signed integer format.

- If the value is greater than 2^31-1 the result saturates to 2^31-1.
- If the value is less than -2^31 the result saturates to -2^31.

**Special Registers Altered:**

`SAT`
Vector Pack Signed Doubleword
Unsigned Saturate VX-form

vpksdus VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
    VR[VRT].word[i] ← Chop| Clamp| ExtendSign(src.dword[i]), 0, 232-1, 32 )
end

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.
The signed integer value in doubleword element i of src is placed into word element i of VR[VRT] in unsigned integer format.
- If the value is greater than 2^32-1 the result saturates to 2^32-1.
- If the value is less than 0 the result saturates to 0.

Special Registers Altered:
SAT

Vector Pack Signed Halfword Signed Saturate VX-form

vpkshss VRT, VRA, VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

do i=0 to 63 by 8
    src1 ← EXTS((VRA)[i×2:i×2+15])
    src2 ← EXTS((VRB)[i×2:i×2+15])
    VRT[i×2:i+7] ← Clamp(src1, -128, 127)24:31
    VRT[i×2+64:i+71] ← Clamp(src2, -128, 127)24:31
end

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.
Signed-integer halfword element i in the source vector is converted to an signed-integer byte.

- If the value of the element is greater than 127 the result saturates to 127
- If the value of the element is less than -128 the result saturates to -128.

The low-order 8 bits of the result is placed into byte element i of VRT.

Special Registers Altered:
SAT
Vector Pack Signed Halfword Unsaturated VX-form

\text{vpkshus} \quad \text{VRT, VRA, VRB}

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 270 \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item do i = 0 to 63 by 8
\item \( \text{src1} \leftarrow \text{EXTS}((\text{VRA})_{i \times 2 : i \times 2 + 15}) \)
\item \( \text{src2} \leftarrow \text{EXTS}((\text{VRB})_{i \times 2 : i \times 2 + 15}) \)
\item \( \text{VRT}_{i : i + 7} \leftarrow \text{Clamp}(\text{src1}, 0, 255)_{24:31} \)
\item \( \text{VRT}_{i + 64 : i + 71} \leftarrow \text{Clamp}(\text{src2}, 0, 255)_{24:31} \)
\end{itemize}

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 15, do the following.
Signed-integer halfword element \( i \) in the source vector is converted to an unsigned-integer byte.
\begin{itemize}
\item If the value of the element is greater than 255, the result saturates to 255
\item If the value of the element is less than 0, the result saturates to 0.
\end{itemize}

The low-order 8 bits of the result is placed into byte element \( i \) of VRT.

Special Registers Altered:
\text{SAT}

Vector Pack Signed Word Signed Saturated VX-form

\text{vpkswss} \quad \text{VRT, VRA, VRB}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 11 & 16 & 21 & 462 \\
\hline
\end{tabular}
\end{center}

\begin{itemize}
\item do i = 0 to 63 by 16
\item \( \text{src1} \leftarrow \text{EXTS}((\text{VRA})_{i \times 2 : i \times 2 + 31}) \)
\item \( \text{src2} \leftarrow \text{EXTS}((\text{VRB})_{i \times 2 : i \times 2 + 31}) \)
\item \( \text{VRT}_{i : i + 15} \leftarrow \text{Clamp}(\text{src1}, -2^{15}, 2^{15} - 1)_{16:31} \)
\item \( \text{VRT}_{i + 64 : i + 79} \leftarrow \text{Clamp}(\text{src2}, -2^{15}, 2^{15} - 1)_{16:31} \)
\end{itemize}

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 7, do the following.
Signed-integer word element \( i \) in the source vector is converted to a signed-integer halfword.
\begin{itemize}
\item If the value of the element is greater than \( 2^{15} - 1 \), the result saturates to \( 2^{15} - 1 \)
\item If the value of the element is less than \( -2^{15} \), the result saturates to \( -2^{15} \).
\end{itemize}

The low-order 16 bits of the result is placed into halfword element \( i \) of VRT.

Special Registers Altered:
\text{SAT}
**Vector Pack Signed Word Unsigned Saturate VX-form**

\[ \text{vpkswus} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>334</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do } i = 0 \text{ to } 63 \text{ by } 16
\]
\[
\begin{align*}
\text{src1} & \leftarrow \text{EXTS}((\text{VRA})_{i×2:i×2+31}) \\
\text{src2} & \leftarrow \text{EXTS}((\text{VRB})_{i×2:i×2+31}) \\
\text{VRT}_{i:31} & \leftarrow \text{Clamp}(\text{src1}, 0, 2^{16} - 1)_{16:31} \\
\text{VRT}_{32:i+79} & \leftarrow \text{Clamp}(\text{src2}, 0, 2^{16} - 1)_{16:31}
\end{align*}
\]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 7, do the following.

Signed-integer word element \( i \) in the source vector is converted to an unsigned-integer halfword.

- If the value of the element is greater than \( 2^{16} - 1 \) the result saturates to \( 2^{16} - 1 \)
- If the value of the element is less than 0 the result saturates to 0.

The low-order 16 bits of the result is placed into halfword element \( i \) of VRT.

Special Registers Altered:

SAT

---

**Vector Pack Unsigned Doubleword Unsigned Saturate VX-form**

\[ \text{vpkudus} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>1230</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{if MSR.VEC then Vector_Unavailable()}
\]
\[
\begin{align*}
\text{src}.\text{qword}[0] & \leftarrow \text{VR}[\text{VRA}] \\
\text{src}.\text{qword}[1] & \leftarrow \text{VR}[\text{VRB}] \\
\text{do } i = 0 \text{ to } 3
\end{align*}
\]
\[
\begin{align*}
\text{VR}[\text{VRT}].\text{word}[i] & \leftarrow \text{Chop}(\text{ExtendZero}(\text{src}.\text{dword}[i]), 0, 2^{32} - 1), 32)
\end{align*}
\]

Let doubleword elements 0 and 1 of \( \text{src} \) be the contents of \( \text{VR}[\text{VRA}] \).

Let doubleword elements 2 and 3 of \( \text{src} \) be the contents of \( \text{VR}[\text{VRB}] \).

For each integer value \( i \) from 0 to 3, do the following.

The unsigned integer value in doubleword element \( i \) of \( \text{src} \) is placed into word element \( i \) of \( \text{VR}[\text{VRT}] \) in unsigned integer format.

- If the value of the element is greater than \( 2^{32} - 1 \) the result saturates to \( 2^{32} - 1 \)

Special Registers Altered:

SAT

---

**Vector Pack Unsigned Halfword Unsigned Modulo VX-form**

\[ \text{vpkuhum} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{if MSR.VEC then Vector_Unavailable()}
\end{align*}
\]
\[
\begin{align*}
\text{src}.\text{qword}[0] & \leftarrow \text{VR}[\text{VRA}] \\
\text{src}.\text{qword}[1] & \leftarrow \text{VR}[\text{VRB}] \\
\text{do } i = 0 \text{ to } 3
\end{align*}
\]
\[
\begin{align*}
\text{VR}[\text{VRT}].\text{word}[i] & \leftarrow \text{Chop}(\text{ExtendZero}(\text{src}.\text{dword}[i]), 32)
\end{align*}
\]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \( i \) from 0 to 15, do the following.

The contents of bits 32:63 of doubleword element \( i \) of \( \text{src} \) is placed into word element \( i \) of \( \text{VR}[\text{VRT}] \).

Special Registers Altered:

None
**Vector Pack Unsigned Halfword Saturate VX-form**

\[
\text{vpkuhus VRT,VRA,VRB}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>15</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
do \text{ i=0 to 63 by 8}
\]

\[
\begin{align*}
\text{src1} & \leftarrow \text{EXTZ}((\text{VRA})_{i \times 2:i \times 2+15}) \\
\text{src2} & \leftarrow \text{EXTZ}((\text{VRB})_{i \times 2:i \times 2+15}) \\
\text{VRT}_{i+7} & \leftarrow \text{Clamp}(\text{src1}, 0, 255)_{24:31} \\
\text{VRT}_{i+64:i+71} & \leftarrow \text{Clamp}(\text{src2}, 0, 255)_{24:31}
\end{align*}
\]

End

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.

Unsigned-integer halfword element i in the source vector is converted to an unsigned-integer byte.

- If the value of the element is greater than 255 the result saturates to 255.

The low-order 8 bits of the result is placed into byte element i of VRT.

**Special Registers Altered:**
- SAT

---

**Vector Pack Unsigned Word Saturate VX-form**

\[
\text{vpkuwus VRT,VRA,VRB}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>15</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
do \text{ i=0 to 63 by 16}
\]

\[
\begin{align*}
\text{src1} & \leftarrow \text{EXTZ}((\text{VRA})_{i \times 2:i \times 2+31}) \\
\text{src2} & \leftarrow \text{EXTZ}((\text{VRB})_{i \times 2:i \times 2+31}) \\
\text{VRT}_{i:i+15} & \leftarrow \text{Clamp}(\text{src1}, 0, 2^{16}-1)_{16:31} \\
\text{VRT}_{i+64:i+79} & \leftarrow \text{Clamp}(\text{src2}, 0, 2^{16}-1)_{16:31}
\end{align*}
\]

End

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.

Unsigned-integer word element i in the source vector is converted to an unsigned-integer halfword.

- If the value of the element is greater than \(2^{16}-1\) the result saturates to \(2^{16}-1\).

The low-order 16 bits of the result is placed into halfword element i of VRT.

**Special Registers Altered:**
- None
Vector Unpack High Pixel VX-form

\[ \text{vupkhpx VRT,VRB} \]

\[
\begin{array}{c|cccc|cccc}
4 & 846 & 11 & 16 & 21 & 24 & 28 & 31 \\
0 & & & & & & & \\
end
\]

do i=0 to 63 by 16
\[
\begin{align*}
\text{VRT}_{i+1:i+7} & \leftarrow \text{EXTS}((\text{VRB})_{i+6}) \\
\text{VRT}_{i+8:i+15} & \leftarrow \text{EXTS}((\text{VRB})_{i+7:i+13}) \\
\text{VRT}_{i+16:i+23} & \leftarrow \text{EXTS}((\text{VRB})_{i+14:i+20}) \\
\text{VRT}_{i+24:i+31} & \leftarrow \text{EXTS}((\text{VRB})_{i+21:i+27})
\end{align*}
\]
end

For each vector element \( i \) from 0 to 3, do the following.
Halfword element \( i \) in VRB is unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element \( i \) of VRT.

Special Registers Altered:
None

Programming Note

The source and target elements can be considered to be 16-bit and 32-bit “pixels” respectively, having the formats described in the Programming Note for the Vector Pack Pixel instruction on page 187.

Vector Unpack Low Pixel VX-form

\[ \text{vupklpx VRT,VRB} \]

\[
\begin{array}{c|cccc|cccc}
4 & 974 & 11 & 16 & 21 & 24 & 28 & 31 \\
5 & & & & & & & \\
end
\]

do i=0 to 63 by 16
\[
\begin{align*}
\text{VRT}_{i+1:i+7} & \leftarrow \text{EXTS}((\text{VRB})_{i+4+64}) \\
\text{VRT}_{i+8:i+15} & \leftarrow \text{EXTS}((\text{VRB})_{i+4+65:i+70}) \\
\text{VRT}_{i+16:i+23} & \leftarrow \text{EXTS}((\text{VRB})_{i+4+71:i+78}) \\
\text{VRT}_{i+24:i+31} & \leftarrow \text{EXTS}((\text{VRB})_{i+4+79:i+86})
\end{align*}
\]
end

For each vector element \( i \) from 0 to 3, do the following.
Halfword element \( i+4 \) in VRB is unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element \( i \) of VRT.

Special Registers Altered:
None

Programming Note

Notice that the unpacking done by the Vector Unpack Pixel instructions does not reverse the packing done by the Vector Pack Pixel instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, Vector Unpack Pixel inserts high-order bits while Vector Pack Pixel discards low-order bits).
### Vector Unpack High Signed Byte VX-form

**vupkhsb**

**VVT,VRB**

$$\text{do } i=0 \text{ to } 63 \text{ by } 8$$

$$\text{VRT}_{i×2:i×2+15} ← \text{EXTS}(\text{VRB}_{i:i+7})$$

**end**

For each vector element \(i\) from 0 to 7, do the following.

Signed-integer byte element \(i\) in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element \(i\) in VRT.

**Special Registers Altered:**

None

### Vector Unpack Low Signed Byte VX-form

**vupklbsb**

**VVT,VRB**

$$\text{do } i=0 \text{ to } 63 \text{ by } 8$$

$$\text{VRT}_{i×2:i×2+15} ← \text{EXTS}(\text{VRB}_{i+8:i+15})$$

**end**

For each vector element \(i\) from 0 to 7, do the following.

Signed-integer byte element \(i+8\) in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element \(i\) in VRT.

**Special Registers Altered:**

None

### Vector Unpack High Signed Halfword VX-form

**vupkhsh**

**VVT,VRB**

$$\text{do } i=0 \text{ to } 63 \text{ by } 16$$

$$\text{VRT}_{i×2:i×2+31} ← \text{EXTS}(\text{VRB}_{i:i+15})$$

**end**

For each vector element \(i\) from 0 to 3, do the following.

Signed-integer halfword element \(i\) in VRB is sign-extended to produce a signed-integer word and placed into word element \(i\) in VRT.

**Special Registers Altered:**

None

### Vector Unpack Low Signed Halfword VX-form

**vupklsh**

**VVT,VRB**

$$\text{do } i=0 \text{ to } 63 \text{ by } 16$$

$$\text{VRT}_{i×2:i×2+31} ← \text{EXTS}(\text{VRB}_{i+4:i+15})$$

**end**

For each vector element \(i\) from 0 to 3, do the following.

Signed-integer halfword element \(i+4\) in VRB is sign-extended to produce a signed-integer word and placed into word element \(i\) in VRT.

**Special Registers Altered:**

None

### Vector Unpack High Signed Word VX-form

**vupkhsw**

**VVT,VRB**

$$\text{VR[VRT].dword[0]} ← \text{Chop}(\text{ExtendSign(}\text{VR[VRB].word[0]}), 64)$$

$$\text{VR[VRT].dword[1]} ← \text{Chop}(\text{ExtendSign(}\text{VR[VRB].word[1]}), 64)$$

For each integer value \(i\) from 0 to 1, do the following.

The signed integer value in word element \(i\) of VRB is sign-extended and placed into doubleword element \(i\) of VR[VRT].

**Special Registers Altered:**

None

### Vector Unpack Low Signed Word VX-form

**vupklsw**

**VVT,VRB**

$$\text{VR[VRT].dword[0]} ← \text{Chop}(\text{ExtendSign(}\text{VR[VRB].word[2]}), 64)$$

$$\text{VR[VRT].dword[1]} ← \text{Chop}(\text{ExtendSign(}\text{VR[VRB].word[3]}), 64)$$

For each integer value \(i\) from 0 to 1, do the following.

The signed integer value in word element \(i+2\) of VRB is sign-extended and placed into doubleword element \(i\) of VR[VRT].

**Special Registers Altered:**

None
### 5.8.2 Vector Merge Instructions

#### Vector Merge High Byte VX-form

**VMRGHb**  

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 7, do the following.  
Byte element \( i \) in VRA is placed into byte element \( 2 \times i \) in VRT.  
Byte element \( i \) in VRB is placed into byte element \( 2 \times i + 1 \) in VRT.

**Special Registers Altered:**  
None

#### Vector Merge Low Byte VX-form

**VMRGLb**  

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 7, do the following.  
Byte element \( i + 8 \) in VRA is placed into byte element \( 2 \times i \) in VRT.  
Byte element \( i + 8 \) in VRB is placed into byte element \( 2 \times i + 1 \) in VRT.

**Special Registers Altered:**  
None

#### Vector Merge High Halfword VX-form

**VMRGHh**  

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 3, do the following.  
Halfword element \( i \) in VRA is placed into halfword element \( 2 \times i \) in VRT.  
Halfword element \( i \) in VRB is placed into halfword element \( 2 \times i + 1 \) in VRT.

**Special Registers Altered:**  
None

#### Vector Merge Low Halfword VX-form

**VMRGLh**  

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

For each vector element \( i \) from 0 to 3, do the following.  
Halfword element \( i + 4 \) in VRA is placed into halfword element \( 2 \times i \) in VRT.  
Halfword element \( i + 4 \) in VRB is placed into halfword element \( 2 \times i + 1 \) in VRT.

**Special Registers Altered:**  
None
Vector Merge High Word VX-form

vmrghw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>140</th>
</tr>
</thead>
</table>

\[
\text{do } i = 0 \text{ to } 63 \text{ by } 32 \\
\text{VRT}_{i\times2:i\times2+31} \leftarrow (\text{VRA})_{i:i+31} \\
\text{VRT}_{i\times2+32:i\times2+63} \leftarrow (\text{VRB})_{i:i+31}
\]

For each vector element \(i\) from 0 to 1, do the following.

Word element \(i\) in VRA is placed into word element \(2i\) in VRT.

Word element \(i\) in VRB is placed into word element \(2i+1\) in VRT.

The word elements in the high-order half of VRA are placed, in the same order, into the even-numbered word elements of VRT. The word elements in the high-order half of VRB are placed, in the same order, into the odd-numbered word elements of VRT.

Special Registers Altered:

None

Vector Merge Low Word VX-form

vmrglw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>0</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>396</th>
</tr>
</thead>
</table>

\[
\text{do } i = 0 \text{ to } 63 \text{ by } 32 \\
\text{VRT}_{i\times2+64:i\times2+95} \leftarrow (\text{VRA})_{i+64:i+95} \\
\text{VRT}_{i\times2+96:i\times2+127} \leftarrow (\text{VRB})_{i:i+31}
\]

For each vector element \(i\) from 0 to 1, do the following.

Word element \(i+2\) in VRA is placed into word element \(2i\) in VRT.

Word element \(i+2\) in VRB is placed into word element \(2i+1\) in VRT.

Special Registers Altered:

None
Vector Merge Even Word VX-form
[Category: Vector-Scalar]

vmrgew VRT, VRA, VRB

\[
\begin{array}{cccccc}
 0 & 6 & 11 & 16 & 21 & 1932 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()
VR[VRT].word[0] ← VR[VRA].word[0]
VR[VRT].word[3] ← VR[VRB].word[0]

The contents of word element 0 of VR[VRA] are placed into word element 0 of VR[VRT].
The contents of word element 0 of VR[VRB] are placed into word element 1 of VR[VRT].
The contents of word element 2 of VR[VRA] are placed into word element 2 of VR[VRT].
The contents of word element 2 of VR[VRB] are placed into word element 3 of VR[VRT].

vmrgew is treated as a Vector instruction in terms of resource availability.

Special Registers Altered
None

Vector Merge Odd Word VX-form
[Category: Vector-Scalar]

vmrgow VRT, VRA, VRB

\[
\begin{array}{cccccc}
 4 & 6 & 11 & 16 & 21 & 1676 \\
\end{array}
\]

if MSR.VEC=0 then Vector_Unavailable()
VR[VRT].word[0] ← VR[VRA].word[1]

The contents of word element 1 of VR[VRA] are placed into word element 0 of VR[VRT].
The contents of word element 1 of VR[VRB] are placed into word element 1 of VR[VRT].
The contents of word element 3 of VR[VRA] are placed into word element 2 of VR[VRT].
The contents of word element 3 of VR[VRB] are placed into word element 3 of VR[VRT].

vmrgow is treated as a Vector instruction in terms of resource availability.

Special Registers Altered
None
5.8.3 Vector Splat Instructions

**Programming Note**
The Vector Splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (e.g., multiplying all elements of a Vector Register by a constant).

### Vector Splat Byte VX-form

#### vspltb

\[ \text{VRT}, \text{VRB}, \text{UIM} \]

\[
\begin{array}{c|c|c|c|c}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{b} \leftarrow \text{UIM} \| 0b0000 \\text{do i=0 to 127 by 8} \\text{VRT}_{i:i+7} \leftarrow (\text{VRB})_{b:b+7} \\text{end}
\]

For each integer value \( i \) from 0 to 15, do the following. The contents of byte element UIM in VRB are placed into byte element \( i \) of VRT.

**Special Registers Altered:**

None

### Vector Splat Halfword VX-form

#### vsplth

\[ \text{VRT}, \text{VRB}, \text{UIM} \]

\[
\begin{array}{c|c|c|c|c}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{b} \leftarrow \text{UIM} \| 0b00000 \\text{do i=0 to 127 by 16} \\text{VRT}_{i:i+15} \leftarrow (\text{VRB})_{b:b+15} \\text{end}
\]

For each integer value \( i \) from 0 to 15, do the following. The contents of halfword element UIM in VRB are placed into halfword element \( i \) of VRT.

**Special Registers Altered:**

None

### Vector Splat Word VX-form

#### vspltw

\[ \text{VRT}, \text{VRB}, \text{UIM} \]

\[
\begin{array}{c|c|c|c|c}
\hline
4 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

\[
\text{b} \leftarrow \text{UIM} \| 0b00000 \\text{do i=0 to 127 by 32} \\text{VRT}_{i:i+31} \leftarrow (\text{VRB})_{b:b+31} \\text{end}
\]

For each integer value \( i \) from 0 to 3, do the following. The contents of word element UIM in VRB are placed into word element \( i \) of VRT.

**Special Registers Altered:**

None
### Vector Splat Immediate Signed Byte VX-form

<table>
<thead>
<tr>
<th>V</th>
<th>4</th>
<th>VRT</th>
<th>SIM</th>
<th>///</th>
<th>780</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

For each integer value \(i\) from 0 to 15, do the following.
The value of the SIM field, sign-extended to 8 bits, is placed into byte element \(i\) of VRT.

**Special Registers Altered:**
None

### Vector Splat Immediate Signed Halfword VX-form

<table>
<thead>
<tr>
<th>V</th>
<th>4</th>
<th>VRT</th>
<th>SIM</th>
<th>///</th>
<th>844</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

For each integer value \(i\) from 0 to 7, do the following.
The value of the SIM field, sign-extended to 16 bits, is placed into halfword element \(i\) of VRT.

**Special Registers Altered:**
None

### Vector Splat Immediate Signed Word VX-form

<table>
<thead>
<tr>
<th>V</th>
<th>4</th>
<th>VRT</th>
<th>SIM</th>
<th>///</th>
<th>908</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

For each vector element \(i\) from 0 to 3, do the following.
The value of the SIM field, sign-extended to 32 bits, is placed into word element \(i\) of VRT.

**Special Registers Altered:**
None

### 5.8.4 Vector Permute Instruction

The **Vector Permute** instruction allows any byte in two source Vector Registers to be copied to any byte in the target Vector Register. The bytes in a third source Vector Register specify from which byte in the first two source Vector Registers the corresponding target byte is to be copied. The contents of the third source Vector Register are sometimes referred to as the “permute control vector”.

#### Vector Permute VA-form

<table>
<thead>
<tr>
<th>V</th>
<th>4</th>
<th>VRT, VRA, VRB, VRC</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>///</td>
<td>43</td>
</tr>
</tbody>
</table>

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value \(i\) from 0 to 15, do the following.
The contents of the byte element in the source vector specified by bits 3:7 of byte element \(i\) of VRC are placed into byte element \(i\) of VRT.

**Special Registers Altered:**
None

**Programming Note**

See the Programming Notes with the **Load Vector for Shift Left** and **Load Vector for Shift Right** instructions on page 186 for examples of uses of vperm.
5.8.5 Vector Select Instruction

**Vector Select VA-form**

```
vsel VRT,VRA,VRB,VRC
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>0</td>
<td>26</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{do } i=0 \text{ to 127}\]

\[\text{VRT}_i \leftarrow (\text{VRC}_i=0) \ ? \ (\text{VRA})_i : (\text{VRB})_i\]

For each bit in VRC that contains the value 0, the corresponding bit in VRA is placed into the corresponding bit of VRT. Otherwise, the corresponding bit in VRB is placed into the corresponding bit of VRT.

**Special Registers Altered:**

None
5.8.6 Vector Shift Instructions

The Vector Shift instructions rotate or shift the contents of a Vector Register or a pair of Vector Registers left or right by a specified number of bytes (vslo, vsro, vsldoi) or bits (vsl, vsr). Depending on the instruction, this “shift count” is specified either by the contents of a Vector Register or by an immediate field in the instruction. In the former case, 7 bits of the shift count register give the shift count in bits (0 ≤ count ≤ 127). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by vslo and vsro; the low-order 3 bits give the number of remaining bits by which to shift and are used by vsl and vsr.

### Vector Shift Left VX-form

**vsl**

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>452</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>452</td>
</tr>
</tbody>
</table>

\[ sh \leftarrow (\text{VRB})_{125:127} \]
\[ t \leftarrow 1 \]
\[ \text{do } i=0 \text{ to } 127 \text{ by } 8 \]
\[ t \leftarrow t \& ((\text{VRB})_{i:7}=sh) \]
\[ \text{if } t=1 \text{ then } \text{VRT} \leftarrow (\text{VRA}) \ll sh \]
\[ \text{else } \text{VRT} \leftarrow \text{undefined} \]

The contents of VRA are shifted left by the number of bits specified in (VRB)_125:127.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into VRT, except if, for any byte element in register VRB, the low-order 3 bits are not equal to the shift amount, then VRT is undefined.

**Special Registers Altered:**

None

### Vector Shift Left Double by Octet Immediate VA-form

**vsldoi**

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>44</td>
</tr>
</tbody>
</table>

\[ \text{VRT} \leftarrow (\text{VRA})_0 \mid (\text{VRB})_8:8+127 \]

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB. Bytes SHB:SHB+15 of the source vector are placed into VRT.

**Special Registers Altered:**

None

### Vector Shift Left by Octet VX-form

**vslo**

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1036</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>1036</td>
</tr>
</tbody>
</table>

\[ shb \leftarrow (\text{VRB})_{121:124} \]
\[ \text{VRT} \leftarrow (\text{VRA}) \ll (shb \mid 0b000) \]

The contents of VRA are shifted left by the number of bytes specified in (VRB)_121:124.

- Bytes shifted out of byte 0 are lost.
- Zeros are supplied to the vacated bytes on the right.

The result is placed into VRT.

**Special Registers Altered:**

None
Vector Shift Right VX-form

vrsh VRT, VRA, VRB

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & VRT & VRA & VRB & 708 \\
\hline
 0 & 4 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

\[
sh \leftarrow (\text{VRB})_{125:127}
\]
\[
t \leftarrow 1
\]
do i=0 to 127 by 8
\[
t \leftarrow t \& (\text{VRB})_{15,16}=sh
\]
end
if t=1 then VRT \leftarrow (VRA) >>ui sh
else VRT undefined

The contents of VRA are shifted right by the number of bits specified in (VRB)_{125:127}.

- Bits shifted out of bit 127 are lost.
- Zeros are supplied to the vacated bits on the left.

The result is placed into VRT, except if, for any byte element in register VRB, the low-order 3 bits are not equal to the shift amount, then VRT is undefined.

Special Registers Altered:
None

Programming Note

A double-register shift by a dynamically specified number of bits (0-127) can be performed in six instructions. The following example shifts Vw || Vx left by the number of bits specified in Vy and places the high-order 128 bits of the result into Vz.

\[
\begin{align*}
\text{vslo} & \quad \text{Vt1.Vw.Vy} \quad \#\text{shift high-order reg left} \\
\text{vsl} & \quad \text{Vt1.Vt1.Vy} \\
\text{vsabkh} & \quad \text{Vt3.V0.Vy} \quad \#\text{adjust shift count (V0)=0} \\
\text{vsro} & \quad \text{Vt2.Vx.Vz} \quad \#\text{shift low-order reg right} \\
\text{vsr} & \quad \text{Vt2.Vt2.Vz} \\
\text{vor} & \quad \text{Vz.Vt1.Vt3} \quad \#\text{merge to get final result}
\end{align*}
\]

Vector Shift Right by Octet VX-form

vsro VRT, VRA, VRB

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & VRT & VRA & VRB & 1100 \\
\hline
 0 & 4 & 6 & 11 & 16 & 21 & 31 \\
\hline
\end{array}
\]

\[
shb \leftarrow (\text{VRB})_{121:124}
\]
\[
\text{VRT} \leftarrow (\text{VRA}) >>ui (shb \| 0b000)
\]

The contents of VRA are shifted right by the number of bytes specified in (VRB)_{121:124}.

- Bytes shifted out of byte 15 are lost.
- Zeros are supplied to the vacated bytes on the left.

The result is placed into VRT.

Special Registers Altered:
None
5.9 Vector Integer Instructions

5.9.1 Vector Integer Arithmetic Instructions

5.9.1.1 Vector Integer Add Instructions

**Vector Add and Write Carry-Out Unsigned Word VX-form**

vaddcuw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element i of VRT.

**Special Registers Altered:**
None

**Vector Add Signed Byte Saturate VX-form**

vaddsbs VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>28</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is added to signed-integer byte element i in VRB.
– If the sum is greater than 127 the result saturates to 127.
– If the sum is less than -128 the result saturates to -128.
The low-order 8 bits of the result are placed into byte element i of VRT.

**Special Registers Altered:**
SAT

**Vector Add Signed Halfword Saturate VX-form**

vaddshs VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is added to signed-integer halfword element i in VRB.
– If the sum is greater than 2^{15}-1 the result saturates to 2^{15}-1
– If the sum is less than -2^{15} the result saturates to -2^{15}.
The low-order 16 bits of the result are placed into halfword element i of VRT.

**Special Registers Altered:**
SAT
**Vector Add Signed Word Saturate**

**VX-form**

\[ vaddsws \]

\[ \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>11</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>21</th>
<th>896</th>
</tr>
</thead>
</table>

\[ \text{do } i = 0 \text{ to } 127 \text{ by } 32 \]

\[ \text{aop} \leftarrow \text{EXTS}((\text{VRA})_i:i+31) \]

\[ \text{bop} \leftarrow \text{EXTS}((\text{VRB})_i:i+31) \]

\[ \text{VRT}_i:i+31 \leftarrow \text{Clamp}(\text{aop} + \text{int} \ bop, -2^{31}, 2^{31}-1) \]

\[ \text{end} \]

For each integer value \( i \) from 0 to 3, do the following.
Signed-integer word element \( i \) in VRA is added to signed-integer word element \( i \) in VRB.

- If the sum is greater than \( 2^{31}-1 \) the result saturates to \( 2^{31}-1 \).
- If the sum is less than \( -2^{31} \) the result saturates to \( -2^{31} \).

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

**Special Registers Altered:**

SAT

**Vector Add Unsigned Byte Modulo**

**VX-form**

\[ vaddubm \]

\[ \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>11</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>21</th>
<th>192</th>
</tr>
</thead>
</table>

\[ \text{do } i = 0 \text{ to } 127 \text{ by } 8 \]

\[ \text{aop} \leftarrow \text{EXTZ}((\text{VRA})_i:i+7) \]

\[ \text{bop} \leftarrow \text{EXTZ}((\text{VRB})_i:i+7) \]

\[ \text{VRT}_i:i+7 \leftarrow \text{Chop}(\text{aop} + \text{int} \ bop, 8) \]

\[ \text{end} \]

For each integer value \( i \) from 0 to 15, do the following.
Unsigned-integer byte element \( i \) in VRA is added to unsigned-integer byte element \( i \) in VRB.

The low-order 8 bits of the result are placed into byte element \( i \) of VRT.

**Special Registers Altered:**

None

**Programming Note**

\[ vaddubm \] can be used for unsigned or signed-integers.

**Vector Add Unsigned Doubleword Modulo**

**VX-form**

\[ vaddudm \]

\[ \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>11</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>21</th>
<th>64</th>
</tr>
</thead>
</table>

\[ \text{do } i = 0 \text{ to } 1 \]

\[ \text{aop} \leftarrow \text{VR}[(\text{VRA}).\text{dword}[i]] \]

\[ \text{bop} \leftarrow \text{VR}[(\text{VRB}).\text{dword}[i]] \]

\[ \text{VR}[(\text{VRT}).\text{dword}[i]] \leftarrow \text{Chop}(\text{aop} + \text{int} \ bop, 64) \]

\[ \text{end} \]

For each integer value \( i \) from 0 to 1, do the following.
The integer value in doubleword element \( i \) of \( \text{VR}[\text{VRB}] \) is added to the integer value in doubleword element \( i \) of \( \text{VR}[\text{VRA}] \).

The low-order 64 bits of the result are placed into doubleword element \( i \) of \( \text{VR}[\text{VRT}] \).

**Special Registers Altered:**

None

**Programming Note**

\[ vaddudm \] can be used for signed or unsigned integers.
Vector Add Unsigned Halfword Modulo VX-form

\[ \text{vadduhm } \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>64</th>
</tr>
</thead>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 16 \\
\quad \text{aop } \leftarrow \text{EXTZ( } \text{(VRA)}_{i:i+15} \text{ )} \\
\quad \text{bop } \leftarrow \text{EXTZ( } \text{(VRB)}_{i:i+15} \text{ )} \\
\quad \text{VRT}_{i:i+15} \leftarrow \text{Chop( aop } + \text{int bop, 16 )} \\
\text{end}
\]

For each integer value \( i \) from 0 to 7, do the following.

Unsigned-integer halfword element \( i \) in VRA is added to unsigned-integer halfword element \( i \) in VRB.

The low-order 16 bits of the result are placed into halfword element \( i \) of VRT.

**Special Registers Altered:**
None

**Programming Note**
\text{vadduhm} can be used for unsigned or signed-integers.

Vector Add Unsigned Word Modulo VX-form

\[ \text{vadduwm } \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
<th>128</th>
</tr>
</thead>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\quad \text{aop } \leftarrow \text{EXTZ( } \text{(VRA)}_{i:i+31} \text{ )} \\
\quad \text{bop } \leftarrow \text{EXTZ( } \text{(VRB)}_{i:i+31} \text{ )} \\
\quad \text{temp } \leftarrow \text{aop } + \text{int bop} \\
\quad \text{VRT}_{i:i+31} \leftarrow \text{Chop( aop } + \text{int bop, 32 )} \\
\text{end}
\]

For each integer value \( i \) from 0 to 3, do the following.

Unsigned-integer word element \( i \) in VRA is added to unsigned-integer word element \( i \) in VRB.

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

**Special Registers Altered:**
None

**Programming Note**
\text{vadduwm} can be used for unsigned or signed-integers.
Vector Add Unsigned Byte Saturate

**VX-form**

vaddubs VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB.

- If the sum is greater than 255, the result saturates to 255.

The low-order 8 bits of the result are placed into byte element i of VRT.

**Special Registers Altered:**

SAT

Vector Add Unsigned Halfword Saturate

**VX-form**

vadduhs VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>576</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following.

Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB.

- If the sum is greater than $2^{16} - 1$, the result saturates to $2^{16} - 1$.

The low-order 16 bits of the result are placed into halfword element i of VRT.

**Special Registers Altered:**

SAT

Vector Add Unsigned Word Saturate

**VX-form**

vadduws VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>640</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following.

Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB.

- If the sum is greater than $2^{32} - 1$, the result saturates to $2^{32} - 1$.

The low-order 32 bits of the result are placed into word element i of VRT.

**Special Registers Altered:**

SAT
Vector Add Unsigned Quadword Modulo VX-form

vadduqm VRT, VRA, VRB

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
sum ← EXTZ(src1) + EXTZ(src2)
VR[VRT] ← Chop(EXTZ(sum), 128)

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
src1 and src2 can be signed or unsigned integers.
The rightmost 128 bits of the sum of src1 and src2 are placed into VR[VRT].

Special Registers Altered:
None

Vector Add Extended Unsigned Quadword Modulo VA-form

vaddeuqm VRT, VRA, VRB, VRC

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
cin ← VR[VRC].bit[127]
sum ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)
VR[VRT] ← Chop(EXTZ(sum), 128)

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].
src1 and src2 can be signed or unsigned integers.
The rightmost 128 bits of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:
None

Vector Add & write Carry Unsigned Quadword VX-form

vaddcuq VRT, VRA, VRB

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>320</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
src1 ← VR[VRB]
src2 ← VR[VRB]
cin ← VR[VRC].bit[127]
sum ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)
VR[VRT] ← Chop(EXTZ(sum), 128)

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
src1 and src2 can be signed or unsigned integers.
The carry out of the sum of src1 and src2 is placed into VR[VRT].

Special Registers Altered:
None

Vector Add Extended & write Carry Unsigned Quadword VA-form

vaddecuq VRT, VRA, VRB, VRC

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
cin ← VR[VRC].bit[127]
sum ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)
VR[VRT] ← Chop(EXTZ(sum), 128)

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].
src1 and src2 can be signed or unsigned integers.
The carry out of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:
None
### Programming Note

The Vector Add Unsigned Quadword instructions support efficient wide-integer addition. The following code sequence can be used to implement a 512-bit signed or unsigned add operation.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vadduqm</td>
<td>vS3,vA3,vB3 # bits 384:511 of sum</td>
</tr>
<tr>
<td>vaddcuq</td>
<td>vC3,vA3,vB3 # carry out of bit 384 of sum</td>
</tr>
<tr>
<td>vadddeuqm</td>
<td>vS2,vA2,vB2,vC3 # bits 256:383 of sum</td>
</tr>
<tr>
<td>vadddecuq</td>
<td>vC2,vA2,vB2,vC3 # carry out of bit 256 of sum</td>
</tr>
<tr>
<td>vadddeuqm</td>
<td>vS1,vA1,vB1,vC2 # bits 128:255 of sum</td>
</tr>
<tr>
<td>vadddecuq</td>
<td>vC1,vA1,vB1,vC2 # carry out of bit 128 of sum</td>
</tr>
<tr>
<td>vadddeuqm</td>
<td>vS0,vA0,vB0,vC1 # bits 0:127 of sum</td>
</tr>
</tbody>
</table>
5.9.1.2 Vector Integer Subtract Instructions

**Vector Subtract and Write Carry-Out**  
*Unsigned Word VX-form*

```
vsbcuw VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1408</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
    aop ← EXTZ((VRA)[i:i+31])
    bop ← EXTZ((VRB)[i:i+31])
    temp ← (aop + int ¬bop + int 1) >> 32
    VRT[i:i+31] ← temp & 0x0000_0001
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into word element i of VRT.

**Special Registers Altered:**  
None

---

**Vector Subtract Signed Halfword Saturate**  
*VX-form*

```
vsubshs VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1856</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 16
    aop ← EXTZ((VRA)[i:i+15])
    bop ← EXTZ((VRB)[i:i+15])
    temp ← aop + int ¬bop + int 1
    VRT[i:i+15] ← Clamp(temp, -2^15, 2^15-1)[16:31]
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRB is subtracted from signed-integer halfword element i in VRA.

– If the intermediate result is greater than 2^15-1 the result saturates to 2^15-1.

– If the intermediate result is less than -2^15 the result saturates to -2^15.

The low-order 16 bits of the result are placed into halfword element i of VRT.

**Special Registers Altered:**  
SAT

---

**Vector Subtract Signed Byte Saturate**  
*VX-form*

```
vsubsbs VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1792</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 8
    aop ← EXTZ((VRA)[i:i+7])
    bop ← EXTZ((VRB)[i:i+7])
    temp ← Clamp(aop + int ¬bop + int 1, -128, 127)[24:31]
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRB is subtracted from signed-integer byte element i in VRA.

– If the intermediate result is greater than 127 the result saturates to 127.

– If the intermediate result is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element i of VRT.

**Special Registers Altered:**  
SAT
**Vector Subtract Signed Word Saturate**

**VX-form**

\[ \text{vsubsws VRT,VRA,VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1920</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>1920</td>
</tr>
</tbody>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by 32}
\]
\[
aop \leftarrow \text{EXTS}((\text{VRA})_{i:i+31})
\]
\[
bop \leftarrow \text{EXTS}((\text{VRB})_{i:i+31})
\]
\[
\text{VRT}_{i:i+31} \leftarrow \text{Clamp}(aop +\text{int} -bop +\text{int} 1,-2^{31},2^{31}-1)
\]

For each integer value \(i\) from 0 to 3, do the following.

Signed-integer word element \(i\) in VRB is subtracted from signed-integer word element \(i\) in VRA.

- If the intermediate result is greater than \(2^{31}-1\) the result saturates to \(2^{31}-1\).
- If the intermediate result is less than \(-2^{31}\) the result saturates to \(-2^{31}\).

The low-order 32 bits of the result are placed into word element \(i\) of VRT.

**Special Registers Altered:**

SAT
Vector Subtract Unsigned Byte Modulo VX-form

\[ \text{vsububm} \ VRT, VRA, VRB \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
4 & 1024 & 6 & 11 & 16 \\
\end{array}
\]

For each integer value \( i \) from 0 to 15, do the following.

Unsigned-integer byte element \( i \) in VRB is subtracted from unsigned-integer byte element \( i \) in VRA. The low-order 8 bits of the result are placed into byte element \( i \) of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Halfword Modulo VX-form

\[ \text{vsubuhm} \ VRT, VRA, VRB \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
4 & 1216 & 6 & 11 & 16 \\
\end{array}
\]

For each integer value \( i \) from 0 to 7, do the following.

Unsigned-integer halfword element \( i \) in VRB is subtracted from unsigned-integer halfword element \( i \) in VRA. The low-order 16 bits of the result are placed into halfword element \( i \) of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Doubleword Modulo VX-form

\[ \text{vsubudm} \ VRT, VRA, VRB \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
4 & 1280 & 6 & 11 & 16 \\
\end{array}
\]

For each integer value \( i \) from 0 to 1, do the following.

The integer value in doubleword element \( i \) of VRB is subtracted from the integer value in doubleword element \( i \) of VRA.

The low-order 64 bits of the result are placed into doubleword element \( i \) of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Word Modulo VX-form

\[ \text{vsubuwm} \ VRT, VRA, VRB \]

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 6 & 11 & 16 \\
4 & 1152 & 6 & 11 & 16 \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.

Unsigned-integer word element \( i \) in VRB is subtracted from unsigned-integer word element \( i \) in VRA. The low-order 32 bits of the result are placed into word element \( i \) of VRT.

Special Registers Altered:
None

Programming Note

\text{vsubudm} \ can be used for signed or unsigned integers.
Vector Subtract Unsigned Byte Saturate VX-form

\[ \text{vsububs} \text{ VRT, VRA, VRB} \]

```
do i=0 to 127 by 8
    aop \leftarrow \text{EXTZ}((\text{VRA})_{i:i+7})
    bop \leftarrow \text{EXTZ}((\text{VRB})_{i:i+7})
    \text{VRT}_{i:i+7} \leftarrow \text{Clamp}(\text{aop} + \text{int} - \text{bop} + \text{int} 1, 0, 255)_{24:31}
end
```

For each integer value \( i \) from 0 to 15, do the following.
Unsigned-integer byte element \( i \) in \( \text{VRB} \) is subtracted from unsigned-integer byte element \( i \) in \( \text{VRA} \). If the intermediate result is less than 0 the result saturates to 0. The low-order 8 bits of the result are placed into byte element \( i \) of \( \text{VRT} \).

Special Registers Altered:
SAT

Vector Subtract Unsigned Halfword Saturate VX-form

\[ \text{vsubuhs} \text{ VRT, VRA, VRB} \]

```
do i=0 to 127 by 16
    aop \leftarrow \text{EXTZ}((\text{VRA})_{i:i+15})
    bop \leftarrow \text{EXTZ}((\text{VRB})_{i:i+15})
    \text{VRT}_{i:i+15} \leftarrow \text{Clamp}(\text{aop} + \text{int} - \text{bop} + \text{int} 1, 0, 2^{32}-1)_{16:31}
end
```

For each integer value \( i \) from 0 to 7, do the following.
Unsigned-integer halfword element \( i \) in \( \text{VRB} \) is subtracted from unsigned-integer halfword element \( i \) in \( \text{VRA} \). If the intermediate result is less than 0 the result saturates to 0.
The low-order 16 bits of the result are placed into halfword element \( i \) of \( \text{VRT} \).

Special Registers Altered:
SAT

Vector Subtract Unsigned Word Saturate VX-form

\[ \text{vsubuws} \text{ VRT, VRA, VRB} \]

```
do i=0 to 127 by 32
    aop \leftarrow \text{EXTZ}((\text{VRA})_{i:i+31})
    bop \leftarrow \text{EXTZ}((\text{VRB})_{i:i+31})
    \text{VRT}_{i:i+31} \leftarrow \text{Clamp}(\text{aop} + \text{int} - \text{bop} + \text{int} 1, 0, 2^{64}-1)_{32:63}
end
```

For each integer value \( i \) from 0 to 7, do the following.
Unsigned-integer word element \( i \) in \( \text{VRB} \) is subtracted from unsigned-integer word element \( i \) in \( \text{VRA} \).
– If the intermediate result is less than 0 the result saturates to 0.
The low-order 32 bits of the result are placed into word element \( i \) of \( \text{VRT} \).

Special Registers Altered:
SAT
### Vector Subtract Unsigned Quadword Modulo VX-form

**vsubuqm**  \( VRT, VRA, VRB \)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>1280</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( src1 \leftarrow VR[VRA] \)
- \( src2 \leftarrow VR[VRB] \)
- \( sum \leftarrow EXTZ(src1) + EXTZ(\neg src2) + EXTZ(1) \)
- \( VR[VRT] \leftarrow Chop(sum, 128) \)

Let \( src1 \) be the integer value in \( VR[VRA] \).
Let \( src2 \) be the integer value in \( VR[VRB] \).

\( src1 \) and \( src2 \) can be signed or unsigned integers.

The rightmost 128 bits of the sum of \( src1 \), the one’s complement of \( src2 \), and the value 1 are placed into \( VR[VRT] \).

**Special Registers Altered:**
- None

### Vector Subtract & write Carry Unsigned Quadword VX-form

**vsubcuq**  \( VRT, VRA, VRB \)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1344</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>1344</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( src1 \leftarrow VR[VRA] \)
- \( src2 \leftarrow VR[VRB] \)
- \( sum \leftarrow EXTZ(src1) + EXTZ(\neg src2) + EXTZ(1) \)
- \( VR[VRT] \leftarrow Chop(EXTZ(Chop(sum >> 128, 1)), 128) \)

Let \( src1 \) be the integer value in \( VR[VRA] \).
Let \( src2 \) be the integer value in \( VR[VRB] \).

\( src1 \) and \( src2 \) can be signed or unsigned integers.

The carry out of the sum of \( src1 \), the one’s complement of \( src2 \), and the value 1 is placed into \( VR[VRT] \).

**Special Registers Altered:**
- None

### Vector Subtract Extended Unsigned Quadword VA-form

**vsubeuqm**  \( VRT, VRA, VRB, VRC \)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>62</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( src1 \leftarrow VR[VRA] \)
- \( src2 \leftarrow VR[VRB] \)
- \( cin \leftarrow VR[VRC].bit[127] \)
- \( sum \leftarrow EXTZ(src1) + EXTZ(\neg src2) + EXTZ(cin) \)
- \( VR[VRT] \leftarrow Chop(sum, 128) \)

Let \( src1 \) be the integer value in \( VR[VRA] \).
Let \( src2 \) be the integer value in \( VR[VRB] \).
Let \( cin \) be the integer value in bit 127 of \( VR[VRC] \).

\( src1 \) and \( src2 \) can be signed or unsigned integers.

The rightmost 128 bits of the sum of \( src1 \), the one’s complement of \( src2 \), and \( cin \) are placed into \( VR[VRT] \).

**Special Registers Altered:**
- None

### Vector Subtract Extended & write Carry Unsigned Quadword VA-form

**vsubecuq**  \( VRT, VRA, VRB, VRC \)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>VRC</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>63</td>
</tr>
</tbody>
</table>

- If MSR.VEC=0 then Vector_Unavailable()
- \( src1 \leftarrow VR[VRA] \)
- \( src2 \leftarrow VR[VRB] \)
- \( cin \leftarrow VR[VRC].bit[127] \)
- \( sum \leftarrow EXTZ(src1) + EXTZ(\neg src2) + EXTZ(cin) \)
- \( VR[VRT] \leftarrow Chop(EXTZ(Chop(sum >> 128, 1)), 128) \)

Let \( src1 \) be the integer value in \( VR[VRA] \).
Let \( src2 \) be the integer value in \( VR[VRB] \).
Let \( cin \) be the integer value in bit 127 of \( VR[VRC] \).

\( src1 \) and \( src2 \) can be signed or unsigned integers.

The carry out of the sum of \( src1 \), the one’s complement of \( src2 \), and \( cin \) are placed into \( VR[VRT] \).

**Special Registers Altered:**
- None
The Vector Subtract Unsigned Quadword instructions support efficient wide-integer subtraction. The following code sequence can be used to implement a 512-bit signed or unsigned subtract operation.

```
vsbugm vS3,vA3,vB3  # bits 384:511 of difference
vsbcug vC3,vA3,vB3  # carry out of bit 384 of difference
vsbemg vS2,vA2,vB2,vC3  # bits 256:383 of difference
vsbecuq vC2,vA2,vB2,vC3  # carry out of bit 256 of difference
vsbugm vS1,vA1,vB1,vC2  # bits 128:255 of difference
vsbcuq vC1,vA1,vB1,vC2  # carry out of bit 128 of difference
vsbugm vS0,vA0,vB0,vC1  # bits 0:127 of difference
```
5.9.1.3 Vector Integer Multiply Instructions

**Vector Multiply Even Signed Byte VX-form**

```
vmulesb VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following. Signed-integer byte element i×2 in VRA is multiplied by signed-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

**Vector Multiply Odd Signed Byte VX-form**

```
vmulosb VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following. Signed-integer byte element i×2+1 in VRA is multiplied by signed-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

**Vector Multiply Even Unsigned Byte VX-form**

```
vmuleub VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following. Unsigned-integer byte element i×2 in VRA is multiplied by unsigned-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

**Vector Multiply Odd Unsigned Byte VX-form**

```
vmuloub VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following. Unsigned-integer byte element i×2+1 in VRA is multiplied by unsigned-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None
**Vector Multiply Even Signed Halfword VX-form**

```
vmulesh VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>840</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i=0 to 127 by 32
prod ← EXTS((VRA)i:i+15) ×si EXTS((VRB)i:i+15)
VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Signed-integer halfword element i×2 in VRA is multiplied by signed-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

**Special Registers Altered:**
None

---

**Vector Multiply Odd Signed Halfword VX-form**

```
vmulosh VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>328</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i=0 to 127 by 32
prod ← EXTS((VRA)i+16:i+31) ×si EXTS((VRB)i+16:i+31)
VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Signed-integer halfword element i×2+1 in VRA is multiplied by signed-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

**Special Registers Altered:**
None

---

**Vector Multiply Even Unsigned Halfword VX-form**

```
vmuleuh VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>584</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i=0 to 127 by 32
prod ← EXTZ((VRA)i:i+15) ×ui EXTZ((VRB)i:i+15)
VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer halfword element i×2 in VRA is multiplied by unsigned-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

**Special Registers Altered:**
None

---

**Vector Multiply Odd Unsigned Halfword VX-form**

```
vmulouh VRT,VRA,VRB
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
do i=0 to 127 by 32
prod ← EXTZ((VRA)i+16:i+31) ×ui EXTZ((VRB)i+16:i+31)
VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer halfword element i×2+1 in VRA is multiplied by unsigned-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

**Special Registers Altered:**
None
Vector Multiply Even Signed Word

**VX-form**

\[ \text{VMULSW} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
    src1 ← VR[VRA].word[2×i]
    src2 ← VR[VRB].word[2×i]
    VR[VRT].dword[i] ← src1 ×si src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The signed integer in word element \( 2×i \) of VR[VRA] is multiplied by the signed integer in word element \( 2×i \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

**Special Registers Altered:**

None

Vector Multiply Even Unsigned Word

**VX-form**

\[ \text{VMPLEW} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
    src1 ← VR[VRA].word[2×i]
    src2 ← VR[VRB].word[2×i]
    VR[VRT].dword[i] ← src1 ×ui src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer in word element \( 2×i \) of VR[VRA] is multiplied by the unsigned integer in word element \( 2×i \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

**Special Registers Altered:**

None

Vector Multiply Odd Signed Word

**VX-form**

\[ \text{VMULOHW} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
    src1 ← VR[VRA].word[2×i+1]
    src2 ← VR[VRB].word[2×i+1]
    VR[VRT].dword[i] ← src1 ×si src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The signed integer in word element \( 2×i + 1 \) of VR[VRA] is multiplied by the signed integer in word element \( 2×i + 1 \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

**Special Registers Altered:**

None

Vector Multiply Odd Unsigned Word

**VX-form**

\[ \text{VMULOUW} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```
do i = 0 to 1
    src1 ← VR[VRA].word[2×i+1]
    src2 ← VR[VRB].word[2×i+1]
    VR[VRT].dword[i] ← src1 ×ui src2
end
```

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer in word element \( 2×i + 1 \) of VR[VRA] is multiplied by the unsigned integer in word element \( 2×i + 1 \) of VR[VRB].

The 64-bit product is placed into doubleword element \( i \) of VR[VRT].

**Special Registers Altered:**

None
**Vector Multiply Unsigned Word Modulo**

**VX-form**

vmuluwm VRT,VRA,VRB

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

do i = 0 to 3
    src1 ← VR[VRA].word[i]
    src2 ← VR[VRB].word[i]
    VR[VRT].word[i] ← Chop(src1 ×ui src2, 32)
end

The integer in word element i of VR[VRA] is multiplied by the integer in word element i of VR[VRB].

The least-significant 32 bits of the product are placed into word element i of VR[VRT].

**Special Registers Altered:**

None

**Programming Note**

*vmuluwm* can be used for unsigned or signed integers.
5.9.1.4 Vector Integer Multiply-Add/Sum Instructions

**Vector Multiply-High-Add Signed Halfword Saturate VA-form**

\[
\text{vmhaddshs} \ VRT, VRA, VRB, VRC
\]

\[
\begin{align*}
do \ i &= 0 \text{ to } 127 \text{ by } 16 \\
\text{prod} &\leftarrow \text{EXTS}((VRA)_{i:i+15}) \times \text{EXTS}((VRB)_{i:i+15}) \\
\text{sum} &\leftarrow (\text{prod} >>_{15} 15) + \text{int} \text{ EXT}(VRC)_{i:i+15} \\
VRT_{i:i+15} &\leftarrow \text{Clamp} \text{ sum, } -2^{15}, 2^{15}-1_{16:31}
\end{align*}
\]

For each vector element \(i\) from 0 to 7, do the following.

Signed-integer halfword element \(i\) in \(VRA\) is multiplied by signed-integer halfword element \(i\) in \(VRB\), producing a 32-bit signed-integer product. Bits 0:16 of the product are added to signed-integer halfword element \(i\) in \(VRC\).

- If the intermediate result is greater than \(2^{15}-1\) the result saturates to \(2^{15}-1\).
- If the intermediate result is less than \(-2^{15}\) the result saturates to \(-2^{15}\).

The low-order 16 bits of the result are placed into halfword element \(i\) of \(VRT\).

**Special Registers Altered:**

SAT

**Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form**

\[
\text{vmhraddshs} \ VRT, VRA, VRB, VRC
\]

\[
\begin{align*}
do \ i &= 0 \text{ to } 127 \text{ by } 16 \\
\text{temp} &\leftarrow \text{EXTS}(VRC)_{i:i+15} \\
\text{prod} &\leftarrow \text{EXTS}(VRA)_{i:i+15} \times \text{EXTS}(VRB)_{i:i+15} \\
\text{sum} &\leftarrow ((\text{prod} + \text{int} \text{ 0x0000_4000}) >>_{15} 15) + \text{int} \text{ temp} \\
VRT_{i:i+15} &\leftarrow \text{Clamp} \text{ sum, } -2^{15}, 2^{15}-1_{16:31}
\end{align*}
\]

For each vector element \(i\) from 0 to 7, do the following.

Signed-integer halfword element \(i\) in \(VRA\) is multiplied by signed-integer halfword element \(i\) in \(VRB\), producing a 32-bit signed-integer product. The value \(0x0000_4000\) is added to the product, producing a 32-bit signed-integer sum. Bits 0:16 of the sum are added to signed-integer halfword element \(i\) in \(VRC\).

- If the intermediate result is greater than \(2^{15}-1\) the result saturates to \(2^{15}-1\).
- If the intermediate result is less than \(-2^{15}\) the result saturates to \(-2^{15}\).

The low-order 16 bits of the result are placed into halfword element \(i\) of \(VRT\).

**Special Registers Altered:**

SAT
Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

\[ \text{vmladduhm} \ VRT, \text{VRA}, \text{VRB}, \text{VRC} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 16 \\
\text{prod} \leftarrow \text{EXTZ}((\text{VRA})_{i:i+15}) \times u_i \text{EXTZ}((\text{VRB})_{i:i+15}) \\
\text{sum} \leftarrow \text{Chop}( \text{prod}, 16 ) + \text{int} ((\text{VRC})_{i:i+15}) \\
\text{VRT}_{i:i+15} \leftarrow \text{Chop}( \text{sum}, 16 ) \\
\text{end}
\]

For each integer value \( i \) from 0 to 3, do the following.

Unsigned-integer halfword element \( i \) in \( \text{VRA} \) is multiplied by unsigned-integer halfword element \( i \) in \( \text{VRB} \), producing a 32-bit unsigned-integer product. The low-order 16 bits of the product are added to unsigned-integer halfword element \( i \) in \( \text{VRC} \).

The low-order 16 bits of the sum are placed into halfword element \( i \) of \( \text{VRT} \).

Special Registers Altered:
None

Programming Note

\text{vmladduhm} \ can be used for unsigned or signed-integers.

Vector Multiply-Sum Unsigned Byte Modulo VA-form

\[ \text{vmsumubm} \ VRT, \text{VRA}, \text{VRB}, \text{VRC} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>36</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 32 \\
\text{temp} \leftarrow \text{EXTZ}((\text{VRC})_{i:i+31}) \\
\text{do } j = 0 \text{ to } 31 \text{ by } 8 \\
\text{prod} \leftarrow \text{EXTZ}((\text{VRA})_{i+j:i+j+7}) \times u_i \text{EXTZ}((\text{VRB})_{i+j:i+j+7}) \\
\text{temp} \leftarrow \text{temp} + \text{int} \text{prod} \\
\text{end} \\
\text{VRT}_{i:i+31} \leftarrow \text{Chop}( \text{temp}, 32 ) \\
\text{end}
\]

For each word element in \( \text{VRT} \) the following operations are performed, in the order shown.

– Each of the four unsigned-integer byte elements contained in the corresponding word element of \( \text{VRA} \) is multiplied by the corresponding unsigned-integer byte element in \( \text{VRB} \), producing an unsigned-integer halfword product.

– The sum of these four unsigned-integer halfword products is added to the unsigned-integer word element in \( \text{VRC} \).

– The unsigned-integer word result is placed into the corresponding word element of \( \text{VRT} \).

Special Registers Altered:
None
Vector Multiply-Sum Mixed Byte Modulo VA-form

vmsummbm VRT,VRA,VRB,VRC

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  temp ← (VRC)i:i+31
  do j=0 to 31 by 8
    prod0:15 ← (VRA)i+j:i+j+7 × (VRB)i+j:i+j+7
    temp ← temp +int EXT[prod]
  end
  VRTi:i+31 ← temp
end

For each word element in VRT the following operations are performed, in the order shown.

– Each of the four signed-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-byte element in VRB, producing a signed-integer product.

– The sum of these four signed-integer byte products is added to the signed-integer word element in VRC.

– The signed-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:
None

Vector Multiply-Sum Signed Halfword Modulo VA-form

vmsumshm VRT,VRA,VRB,VRC

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  temp ← (VRC)i:i+31
  do j=0 to 31 by 16
    prod0:31 ← (VRA)i+j:i+j+15 × (VRB)i+j:i+j+15
    temp ← temp +int prod
  end
  VRTi:i+31 ← temp
end

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

– The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

– The signed-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:
None
Vector Multiply-Sum Signed Halfword Saturate VA-form

\[
vmsumshs \quad \text{VRT}, \text{VRA}, \text{VRB}, \text{VRC}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>31</th>
</tr>
</thead>
</table>

\[
\begin{align*}
do \ i = 0 \ to \ 127 \ by \ 32 \\
& \text{temp} \leftarrow \text{EXTS}(\text{VRC}_{i:i+31}) \\
& \text{do} \ j = 0 \ to \ 31 \ by \ 16 \\
& \quad \text{srcA} \leftarrow \text{EXTS}(\text{VRA}_{i+j:i+j+15}) \\
& \quad \text{srcB} \leftarrow \text{EXTS}(\text{VRB}_{i+j:i+j+15}) \\
& \quad \text{prod} \leftarrow \text{srcA} \times \text{srcB} \\
& \quad \text{temp} \leftarrow \text{temp} + \text{int prod} \\
& \text{end} \\
& \text{VRT}_{i:i+31} \leftarrow \text{Clamp}(\text{temp}, -2^{31}, 2^{31}-1) \\
& \text{end}
\end{align*}
\]

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

- The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

- If the intermediate result is greater than \(2^{31}-1\) the result saturates to \(2^{31}-1\) and if it is less than \(-2^{31}\) it saturates to \(-2^{31}\).

- The result is placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

Vector Multiply-Sum Unsigned Halfword Modulo VA-form

\[
vmsumuhm \quad \text{VRT}, \text{VRA}, \text{VRB}, \text{VRC}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>38</th>
<th>31</th>
</tr>
</thead>
</table>

\[
\begin{align*}
do \ i = 0 \ to \ 127 \ by \ 32 \\
& \text{temp} \leftarrow \text{EXTZ}(\text{VRC}_{i:i+31}) \\
& \text{do} \ j = 0 \ to \ 31 \ by \ 16 \\
& \quad \text{srcA} \leftarrow \text{EXTZ}(\text{VRA}_{i+j:i+j+15}) \\
& \quad \text{srcB} \leftarrow \text{EXTZ}(\text{VRB}_{i+j:i+j+15}) \\
& \quad \text{prod} \leftarrow \text{srcA} \times \text{srcB} \\
& \quad \text{temp} \leftarrow \text{temp} + \text{int prod} \\
& \text{end} \\
& \text{VRT}_{i:i+31} \leftarrow \text{Chop}(\text{temp}, 32) \\
& \text{end}
\end{align*}
\]

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer word product.

- The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

- The unsigned-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:

None
Vector Multiply-Sum Unsigned Halfword
Saturate VA-form

\texttt{vmsumuhs} VRT,VRA,VRB,VRC

\begin{verbatim}
  do i=0 to 127 by 32
    temp ← EXTZ((VRC)i:i+31)
    do j=0 to 31 by 16
      src1 ← EXTZ((VRA)i+j:i+j+15)
      src2 ← EXTZ((VRB)i+j:i+j+15)
      prod ← src1 ×ui src2
    end
    temp ← temp +int prod
    VRTi:i+31 ← Clamp(temp, 0, 2^{32}-1)
  end
\end{verbatim}

For each word element in VRT the following operations are performed, in the order shown.

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer product.

- The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

- If the intermediate result is greater than $2^{32}-1$ the result saturates to $2^{32}-1$.

- The result is placed into the corresponding word element of VRT.

\textbf{Special Registers Altered:}

SAT
## 5.9.1.5 Vector Integer Sum-Across Instructions

### Vector Sum across Signed Word Saturate VX-form

**VSUMSWS**, **VRT**, **VRA**, **VRB**

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1928</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
temp ← EXTS((VRB)36:127)
do i=0 to 127 by 32
   temp ← temp +int EXTS((VRA)i:i+31)
end
VRT0:31 ← 0x0000_0000
VRT32:63 ← 0x0000_0000
VRT64:95 ← 0x0000_0000
VRT96:127 ← Clamp(temp, -2^31, 2^31-1)
```

The sum of the four signed-integer word elements in **VRA** is added to signed-integer word element 3 of **VRB**.

- If the intermediate result is greater than 2^31-1 the result saturates to 2^31-1.
- If the intermediate result is less than -2^31 the result saturates to -2^31.

The low-end 32 bits of the result are placed into word element 3 of **VRT**.

Word elements 0 to 2 of **VRT** are set to 0.

**Special Registers Altered:**

SAT

---

### Vector Sum across Half Signed Word Saturate VX-form

**VSUM2WS**, **VRT**, **VRA**, **VRB**

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1672</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
do i=0 to 127 by 64
   temp ← EXTS((VRB)i+32:i+63)
do j=0 to 63 by 32
   temp ← temp +int EXTS((VRA)i+j:i+j+31)
end
VRTi:i+63 ← 0x0000_0000 & Clamp(temp, -2^31, 2^31-1)
```

Word elements 0 and 2 of **VRT** are set to 0.

The sum of the signed-integer word elements 0 and 1 in **VRA** is added to the signed-integer word element in bits 32:63 of **VRB**.

- If the intermediate result is greater than 2^31-1 the result saturates to 2^31-1.
- If the intermediate result is less than -2^31 the result saturates to -2^31.

The low-order 32 bits of the result are placed into word element 1 of **VRT**.

The sum of signed-integer word elements 2 and 3 in **VRA** is added to the signed-integer word element in bits 96:127 of **VRB**.

- If the intermediate result is greater than 2^31-1 the result saturates to 2^31-1.
- If the intermediate result is less than -2^31 the result saturates to -2^31.

The low-order 32 bits of the result are placed into word element 3 of **VRT**.

**Special Registers Altered:**

SAT
**Vector Sum across Quarter Signed Byte Saturate VX-form**

\[
vsum4sbs\begin{array}{c|c|c|c|c}
\text{VRT} & \text{VRA} & \text{VRB} & \text{1800} \\
4 & 6 & 31 & 21 & 31 \\
\end{array}
\]

\begin{verbatim}
do i=0 to 127 by 32
    temp ← EXTS((VRB)_{i:i+31})
do j=0 to 31 by 8
    temp ← temp + int EXTS((VRA)_{i+j:i+j+7})
end
VRT_{i:i+31} ← Clamp(temp, -2^{31}, 2^{31}-1)
end
\end{verbatim}

For each integer value \( i \) from 0 to 3, do the following.

The sum of the four signed-integer byte elements contained in word element \( i \) of VRA is added to signed-integer word element \( i \) in VRB.

- If the intermediate result is greater than \( 2^{31}-1 \) the result saturates to \( 2^{31}-1 \).
- If the intermediate result is less than \( -2^{31} \) the result saturates to \( -2^{31} \).

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

Special Registers Altered:

SAT

**Vector Sum across Quarter Signed Halfword Saturate VX-form**

\[
vsum4shs\begin{array}{c|c|c|c|c}
\text{VRT} & \text{VRA} & \text{VRB} & \text{1608} \\
4 & 6 & 11 & 16 & 31 \\
\end{array}
\]

\begin{verbatim}
do i=0 to 127 by 32
    temp ← EXTS((VRB)_{i:i+31})
do j=0 to 31 by 16
    temp ← temp + int EXTS((VRA)_{i+j:i+j+15})
end
VRT_{i:i+31} ← Clamp(temp, -2^{31}, 2^{31}-1)
end
\end{verbatim}

For each integer value \( i \) from 0 to 3, do the following.

The sum of the two signed-integer halfword elements contained in word element \( i \) of VRA is added to signed-integer word element \( i \) in VRB.

- If the intermediate result is greater than \( 2^{31}-1 \) the result saturates to \( 2^{31}-1 \).
- If the intermediate result is less than \( -2^{31} \) the result saturates to \( -2^{31} \).

The low-order 32 bits of the result are placed into the corresponding word element of VRT.

Special Registers Altered:

SAT
**Vector Sum across Quarter Unsigned**
**Byte Saturate VX-form**

\[ \text{vsum4ubs VRT,VRA,VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1544</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 3, do the following.

The sum of the four unsigned-integer byte elements contained in word element \( i \) of VRA is added to unsigned-integer word element \( i \) in VRB.

- If the intermediate result is greater than \( 2^{32}-1 \) it saturates to \( 2^{32}-1 \).

The low-order 32 bits of the result are placed into word element \( i \) of VRT.

**Special Registers Altered:**

SAT
5.9.1.6 Vector Integer Average Instructions

**Vector Average Signed Byte VX-form**

\[ \text{vavgsb} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1282</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 8
  aop ← EXT((VRA)_i:i+7)
  bop ← EXT((VRB)_i:i+7)
  VRT_i:i+7 ← Chop((aop +int bop +int 1) >> 1, 8)
end

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is added to signed-integer byte element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element i of VRT.

**Special Registers Altered:**
None

**Vector Average Signed Halfword VX-form**

\[ \text{vavgsh} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1346</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 16
  aop ← EXT((VRA)_i:i+15)
  bop ← EXT((VRB)_i:i+15)
  VRT_i:i+15 ← Chop((aop +int bop +int 1) >> 1, 16)
end

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is added to signed-integer halfword element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element i of VRT.

**Special Registers Altered:**
None

**Vector Average Signed Word VX-form**

\[ \text{vavgsw} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1410</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
  aop ← EXT((VRA)_i:i+31)
  bop ← EXT((VRB)_i:i+31)
  VRT_i:i+31 ← Chop((aop +int bop +int 1) >> 1, 32)
end

For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is added to signed-integer word element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element i of VRT.

**Special Registers Altered:**
None
Vector Average Unsigned Byte VX-form

```
vavgub VRT, VRA, VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop((aop + int bop + int 1) >> ui 1, 8)
end

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
None

Vector Average Unsigned Halfword VX-form

```
vavguh VRT, VRA, VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop((aop + int bop + int 1) >> ui 1, 16)
end

For each integer value i from 0 to 7, do the following.

Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Average Unsigned Word VX-form

```
vavguw VRT, VRA, VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>
```

do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop((aop + int bop + int 1) >> ui 1, 32)
end

For each integer value i from 0 to 3, do the following.

Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
None
5.9.1.7 Vector Integer Maximum and Minimum Instructions

### Vector Maximum Signed Byte VX-form

**vmaxsb**<br>**VRT,VRA,VRB**

```
<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>258</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>
```

For each integer value i from 0 to 127 by 8,

```
do i=0 to 127 by 8
  aop ← EXTB(VRA)1:i+7
  bop ← EXTB(VRB)1:i+7
  VRT1:i+7 ← (aop >si bop) ? (VRA)1:i+7 : (VRB)1:i+7
end
```

For each integer value i from 0 to 15, do the following. Signed-integer byte element i in VRA is compared to signed-integer byte element i in VRB. The larger of the two values is placed into byte element i of VRT.

**Special Registers Altered:**
None

### Vector Maximum Unsigned Byte VX-form

**vmaxub**<br>**VRT,VRA,VRB**

```
<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>3</td>
</tr>
</tbody>
</table>
```

For each integer value i from 0 to 127 by 8,

```
do i=0 to 127 by 8
  aop ← EXTZ(VRA)1:i+7
  bop ← EXTZ(VRB)1:i+7
  VRT1:i+7 ← (aop >ui bop) ? (VRA)1:i+7 : (VRB)1:i+7
end
```

For each integer value i from 0 to 15, do the following. Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. The larger of the two values is placed into byte element i of VRT.

**Special Registers Altered:**
None

### Vector Maximum Signed Doubleword VX-form

**vmaxsd**<br>**VRT,VRA,VRB**

```
<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>450</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>
```

For each integer value i from 0 to 1,

```
do i = 0 to 1
  aop ← VR[VRA].dword[i]
  bop ← VR[VRB].dword[i]
  VR[VRT].dword[i] ← (aop >si bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following. The signed integer value in doubleword element i of VR[VRA] is compared to the signed integer value in doubleword element i of VR[VRB]. The larger of the two values is placed into doubleword element i of VR[VRT].

**Special Registers Altered:**
None

### Vector Maximum Unsigned Doubleword VX-form

**vmaxud**<br>**VRT,VRA,VRB**

```
<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>194</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>
```

For each integer value i from 0 to 1,

```
do i = 0 to 1
  aop ← VR[VRA].dword[i]
  bop ← VR[VRB].dword[i]
  VR[VRT].dword[i] ← (aop >ui bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following. The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. The larger of the two values is placed into doubleword element i of VR[VRT].

**Special Registers Altered:**
None
Vector Maximum Signed Halfword VX-form

\texttt{vmaxsh} \hspace{1em} \texttt{VRT, VRA, VRB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 16 & 21 & 32 & 322 \\
\hline
\end{tabular}

\begin{itemize}
\item do i=0 to 127 by 16
\item aop \leftarrow \texttt{EXTS(VRA)}_{i:i+15}
\item bop \leftarrow \texttt{EXTS(VRB)}_{i:i+15}
\item \texttt{VRT} \leftarrow (\texttt{aop} >_{ui} \texttt{bop}) ? (\texttt{VRA})_{i:i+15} : (\texttt{VRB})_{i:i+15}
\end{itemize}

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. The larger of the two values is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Maximum Signed Word VX-form

\texttt{vmaxsw} \hspace{1em} \texttt{VRT, VRA, VRB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 16 & 21 & 386 \\
\hline
\end{tabular}

\begin{itemize}
\item do i=0 to 127 by 32
\item aop \leftarrow \texttt{EXTS(VRA)}_{i:i+31}
\item bop \leftarrow \texttt{EXTS(VRB)}_{i:i+31}
\item \texttt{VRT} \leftarrow (\texttt{aop} >_{ui} \texttt{bop}) ? (\texttt{VRA})_{i:i+31} : (\texttt{VRB})_{i:i+31}
\end{itemize}

For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. The larger of the two values is placed into word element i of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Halfword VX-form

\texttt{vmaxuh} \hspace{1em} \texttt{VRT, VRA, VRB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 16 & 21 & 32 & 66 \\
\hline
\end{tabular}

\begin{itemize}
\item do i=0 to 127 by 16
\item aop \leftarrow \texttt{EXTZ(VRA)}_{i:i+15}
\item bop \leftarrow \texttt{EXTZ(VRB)}_{i:i+15}
\item \texttt{VRT} \leftarrow (\texttt{aop} >_{ui} \texttt{bop}) ? (\texttt{VRA})_{i:i+15} : (\texttt{VRB})_{i:i+15}
\end{itemize}

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. The larger of the two values is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Word VX-form

\texttt{vmaxuw} \hspace{1em} \texttt{VRT, VRA, VRB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & 6 & 16 & 21 & 386 & 130 \\
\hline
\end{tabular}

\begin{itemize}
\item do i=0 to 127 by 32
\item aop \leftarrow \texttt{EXTZ(VRA)}_{i:i+31}
\item bop \leftarrow \texttt{EXTZ(VRB)}_{i:i+31}
\item \texttt{VRT} \leftarrow (\texttt{aop} >_{ui} \texttt{bop}) ? (\texttt{VRA})_{i:i+31} : (\texttt{VRB})_{i:i+31}
\end{itemize}

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. The larger of the two values is placed into word element i of VRT.

Special Registers Altered:
None
**Vector Minimum Signed Byte VX-form**

\[ \text{vmins} \backslash \text{b} \ VRT, VRA, VRB \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>770</th>
</tr>
</thead>
</table>

For each integer value \( i \) from 0 to 15, do the following.

The signed integer byte element \( i \) in VRA is compared to signed integer byte element \( i \) in VRB. The smaller of the two values is placed into byte element \( i \) of VRT.

**Special Registers Altered:**

None

---

**Vector Minimum Unsigned Byte VX-form**

\[ \text{vmin} \backslash \text{ub} \ VRT, VRA, VRB \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>774</th>
</tr>
</thead>
</table>

For each integer value \( i \) from 0 to 127, do the following.

Unsigned integer byte element \( i \) in VRA is compared to unsigned integer byte element \( i \) in VRB. The smaller of the two values is placed into byte element \( i \) of VRT.

**Special Registers Altered:**

None

---

**Vector Minimum Signed Doubleword VX-form**

\[ \text{vmin} \backslash \text{sd} \ VRT, VRA, VRB \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>962</th>
</tr>
</thead>
</table>

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in doubleword element \( i \) of VRA is compared to the signed integer value in doubleword element \( i \) of VRB. The smaller of the two values is placed into doubleword element \( i \) of VRT.

**Special Registers Altered:**

None

---

**Vector Minimum Unsigned Doubleword VX-form**

\[ \text{vmin} \backslash \text{ud} \ VRT, VRA, VRB \]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>706</th>
</tr>
</thead>
</table>

For each integer value \( i \) from 0 to 1, do the following.

The unsigned integer value in doubleword element \( i \) of VRA is compared to the unsigned integer value in doubleword element \( i \) of VRB. The smaller of the two values is placed into doubleword element \( i \) of VRT.

**Special Registers Altered:**

None
Vector Minimum Signed Halfword VX-form

\[ \text{vminsh} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
\begin{verbatim}
do i=0 to 127 by 16
  aop  \leftarrow \text{EXTS}((\text{VRA})_{i:i+15})
  bop  \leftarrow \text{EXTS}((\text{VRB})_{i:i+15})
  \text{VRT}_{i:i+15} \leftarrow (\text{aop} <_{ui} \text{bop}) \cdot (\text{VRA})_{i:i+15} + (\text{VRB})_{i:i+15}
\end{verbatim}
```

For each integer value \( i \) from 0 to 7, do the following.
Signed-integer halfword element \( i \) in VRA is compared to signed-integer halfword element \( i \) in VRB. The smaller of the two values is placed into halfword element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Signed Word VX-form

\[ \text{vminsw} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
\begin{verbatim}
do i=0 to 127 by 32
  aop  \leftarrow \text{EXTS}((\text{VRA})_{i:i+31})
  bop  \leftarrow \text{EXTS}((\text{VRB})_{i:i+31})
  \text{VRT}_{i:i+31} \leftarrow (\text{aop} <_{ui} \text{bop}) \cdot (\text{VRA})_{i:i+31} + (\text{VRB})_{i:i+31}
\end{verbatim}
```

For each integer value \( i \) from 0 to 3, do the following.
Signed-integer word element \( i \) in VRA is compared to signed-integer word element \( i \) in VRB. The smaller of the two values is placed into word element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Halfword VX-form

\[ \text{vminuh} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
\begin{verbatim}
do i=0 to 127 by 16
  aop  \leftarrow \text{EXTZ}((\text{VRA})_{i:i+15})
  bop  \leftarrow \text{EXTZ}((\text{VRB})_{i:i+15})
  \text{VRT}_{i:i+15} \leftarrow (\text{aop} <_{ui} \text{bop}) \cdot (\text{VRA})_{i:i+15} + (\text{VRB})_{i:i+15}
\end{verbatim}
```

For each integer value \( i \) from 0 to 7, do the following.
Unsigned-integer halfword element \( i \) in VRA is compared to unsigned-integer halfword element \( i \) in VRB. The smaller of the two values is placed into halfword element \( i \) of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Word VX-form

\[ \text{vminuw} \quad \text{VRT}, \text{VRA}, \text{VRB} \]

```
\begin{verbatim}
do i=0 to 127 by 32
  aop  \leftarrow \text{EXTZ}((\text{VRA})_{i:i+31})
  bop  \leftarrow \text{EXTZ}((\text{VRB})_{i:i+31})
  \text{VRT}_{i:i+31} \leftarrow (\text{aop} <_{ui} \text{bop}) \cdot (\text{VRA})_{i:i+31} + (\text{VRB})_{i:i+31}
\end{verbatim}
```

For each integer value \( i \) from 0 to 3, do the following.
Unsigned-integer word element \( i \) in VRA is compared to unsigned-integer word element \( i \) in VRB. The smaller of the two values is placed into word element \( i \) of VRT.

Special Registers Altered:
None
5.9.2 Vector Integer Compare Instructions

The **Vector Integer Compare** instructions compare two Vector Registers element by element, interpreting the elements as unsigned or signed-integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and to all 0s if the relation being tested is false.

If \( Rc=1 \) CR Field 6 is set to reflect the result of the comparison, as follows.

**Vector Compare Equal To Unsigned Byte VC-form**

<table>
<thead>
<tr>
<th>Vcmpequb</th>
<th>VRT, VRA, VRB ( (Rc=0) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vcmpequb.</td>
<td>VRT, VRA, VRB ( (Rc=1) )</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 15, do the following.

Unsigned-integer byte element \( i \) in VRA is compared to unsigned-integer byte element \( i \) in VRB. Byte element \( i \) in VRT is set to all 1s if unsigned-integer byte element \( i \) in VRA is equal to unsigned-integer byte element \( i \) in VRB, and is set to all 0s otherwise.

**Special Registers Altered:**

\( CR \) field \( 6 \) .......................... \( (if \ Rc=1) \)

**Vector Compare Equal To Unsigned Halfword VC-form**

<table>
<thead>
<tr>
<th>Vcmpequh</th>
<th>VRT, VRA, VRB ( (Rc=0) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vcmpequh.</td>
<td>VRT, VRA, VRB ( (Rc=1) )</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 7, do the following.

Unsigned-integer halfword element \( i \) in VRA is compared to unsigned-integer halfword element \( i \) in VRB. Halfword element \( i \) in VRT is set to all 1s if unsigned-integer halfword element \( i \) in VRA is equal to unsigned-integer halfword element \( i \) in VRB, and is set to all 0s otherwise.

**Special Registers Altered:**

\( CR \) field \( 6 \) .......................... \( (if \ Rc=1) \)
Vector Compare Equal To Unsigned Word
VC-form

vcmpequw VRT,VRA,VRB  \((Rc=0)\)
vcmpequw. VRT,VRA,VRB  \((Rc=1)\)

\[
\begin{array}{cccccc}
4 & 6 & 11 & 16 & 21 & 31 \\
0 & VRT & VRA & VRB & Rc & 134 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & VRT & VRA & VRB \\
0 & 6 & 11 & 16 & 21 & 22 & 31 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
4 & 6 & 11 & 16 & 21 & 22 & 31 \\
0 & VRT & VRA & VRB & Rc & 199 \\
\end{array}
\]

do i=0 to 127 by 32
VRT(i:i+31) = int (VRB(i:i+31)) \(\cong\) \(\texttt{321}\)
end

if \(Rc=1\) then do
\[
\begin{align*}
\text{do } & i \text{ = 0 to 1} \\
\text{aop } & \leftarrow \text{EXTZ(VR[VRA].dword}[i]) \\
bop & \leftarrow \text{EXTZ(VR[VRB].dword}[i]) \\
\text{if } & (\text{aop} = \text{bop}) \text{ then do} \\
\text{flag.bit}[i] & \leftarrow \text{0b1} \\
\text{end} \\
\text{else do} \\
\text{flag.bit}[i] & \leftarrow \text{0b0} \\
\text{end} \\
\text{end} \\
\text{if } & \text{Rc=1 then do} \\
\text{CR.bit}[24] & \leftarrow (\text{flag}=\text{0b11}) \\
\text{CR.bit}[25] & \leftarrow \text{0b0} \\
\text{CR.bit}[26] & \leftarrow (\text{flag}=\text{0b00}) \\
\text{CR.bit}[27] & \leftarrow \text{0b0} \\
\text{end} \\
\end{align*}
\]

For each integer value \(i\) from 0 to 3, do the following.
The unsigned integer value in word element \(i\) in VR[VRA] is compared to the unsigned integer value in word element \(i\) in VR[VRB]. Word element \(i\) in VR[VRT] is set to all 1s if unsigned-integer word element \(i\) in VR[VRA] is equal to unsigned-integer word element \(i\) in VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:
\[
\text{CR field } 6 \ldots \ldots \ldots (\text{if } Rc=1)
\]
Vector Compare Greater Than Signed Byte VC-form

vcmpgtsb VRT,VRA,VRB \((Rc=0)\)
vcmpgtsb. VRT,VRA,VRB \((Rc=1)\)

<table>
<thead>
<tr>
<th>4</th>
<th>774</th>
</tr>
</thead>
</table>
| 0 | i=0 to 127 by 8
| 6 | VRT, | 7 | VRA, | 16 | VRB, | 21 | 22 | 31 |

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 127 \text{ by } 8 \\
VRT_{i:,i+7} &= \left\{ \begin{array}{l}
\text{if } (\text{VRA})_{i:,i+7} \geq (\text{VRB})_{i:,i+7} \text{ then set } \text{VRT}_{i:,i+7} \text{ to all } 1s \\
\text{else set } \text{VRT}_{i:,i+7} \text{ to all } 0s \\
\end{array} \right. \\
\text{end}
\end{align*}
\]

For each integer value \( i \) from 0 to 15, do the following.

The signed integer value in byte element \( i \) in \( \text{VR[VRT]} \) is compared to the signed integer value in byte element \( i \) in \( \text{VR[VRB]} \). Byte element \( i \) in \( \text{VR[VRT]} \) is set to all 1s if signed-integer byte element \( i \) in \( \text{VR[VRA]} \) is greater than to signed-integer byte element \( i \) in \( \text{VR[VRB]} \), and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . (if \( Rc=1 \))

Vector Compare Greater Than Signed Doubleword VX-form

vcmpgtsd VRT,VRA,VRB \((Rc=0)\)
vcmpgtsd. VRT,VRA,VRB \((Rc=1)\)

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \\
\text{aop } &= \text{EXTS}(\text{VR[VRA]}.\text{dword}[i]) \\
\text{bop } &= \text{EXTS}(\text{VR[VRB]}.\text{dword}[i]) \\
\text{if } (\text{aop} \geq \text{bop}) \text{ then do} \\
\text{VR[VRT].dword}[i] &= 0xFFFF_FFFF_FFFF_FFFF \\
\text{flag.bit}[i] &= 0b1 \\
\text{end} \\
\text{else do} \\
\text{VR[VRT].dword}[i] &= 0x0000_0000_0000_0000 \\
\text{flag.bit}[i] &= 0b0 \\
\text{end} \\
\text{end}
\end{align*}
\]

For each integer value \( i \) from 0 to 1, do the following.

The signed integer value in doubleword element \( i \) of \( \text{VR[VRA]} \) is compared to the signed integer value in doubleword element \( i \) of \( \text{VR[VRB]} \). Doubleword element \( i \) of \( \text{VR[VRT]} \) is set to all 1s if the signed integer value in doubleword element \( i \) of \( \text{VR[VRA]} \) is greater than the signed integer value in doubleword element \( i \) of \( \text{VR[VRB]} \), and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . (if \( Rc=1 \))
Vector Compare Greater Than Signed
Halfword VC-form

\[
vcmpgtsh \quad VRT, VRA, VRB \quad (Rc=0) 
vcmpgtsh. \quad VRT, VRA, VRB \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>838</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>06</td>
<td>2122</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 7, do the following.
Signed-integer halfword element \( i \) in VRA is compared to signed-integer halfword element \( i \) in VRB. Halfword element \( i \) in VRT is set to all 1s if signed-integer halfword element \( i \) in VRA is greater than signed-integer halfword element \( i \) in VRB, and is set to all 0s otherwise.

Special Registers Altered:
\( CR \) field 6 ...........................................(if Rc=1)

Vector Compare Greater Than Signed
Word VC-form

\[
vcmpgtsw \quad VRT, VRA, VRB \quad (Rc=0) 
vcmpgtsw. \quad VRT, VRA, VRB \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>16</th>
<th>21</th>
<th>32</th>
<th>902</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>06</td>
<td>2132</td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 3, do the following.
Signed-integer word element \( i \) in VRA is compared to signed-integer word element \( i \) in VRB. Word element \( i \) in VRT is set to all 1s if signed-integer word element \( i \) in VRA is greater than signed-integer word element \( i \) in VRB, and is set to all 0s otherwise.

Special Registers Altered:
\( CR \) field 6 ...........................................(if Rc=1)
Vector Compare Greater Than Unsigned Byte VC-form

vcmpgtub VRT, VRA, VRB (Rc=0)
vcmpgtub. VRT, VRA, VRB (Rc=1)

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

do i = 0 to 127 by 8
    VRT_{i:i+7} ← ((VRA)_{i:i+7} >_{ui} (VRB)_{i:i+7}) * 1 : 0
end

if Rc = 1 then do
    t ← VRT_{i:i+7}
    i ← VRT_{i:i+7}
    CR6 ← t || 0b0 || i || 0b0
end

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. Byte element i in VRT is set to all 1s if unsigned-integer byte element i in VRA is greater than to unsigned-integer byte element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 ........................... (if Rc = 1)

Vector Compare Greater Than Unsigned Doubleword VX-form

vcmpgtud VRT, VRA, VRB (Rc=0)
vcmpgtud. VRT, VRA, VRB (Rc=1)

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

do i = 0 to 1
    aop ← EXTZ(VR[VRA].dword[i])
    bop ← EXTZ(VR[VRB].dword[i])
    if (ExtendZero(aop) >_{ui} ExtendZero(bop)) then do
        VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
        flag.bit[i] ← 0b1
    end
    else do
        VR[VRT].dword[i] ← 0x0000_0000_0000_0000
        flag.bit[i] ← 0b1
    end
end

if "vcmpgtud." then do
    CR.bit[24] ← (flag=0b11)
    CR.bit[25] ← 0b0
    CR.bit[26] ← (flag=0b0)
    CR.bit[27] ← 0b0
end

For each integer value i from 0 to 1, do the following.
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the unsigned integer value in doubleword element i of VR[VRA] is greater than the unsigned integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:
CR field 6  ................................... (if Rc = 1)
Vector Compare Greater Than Unsigned Halfword VC-form

```
vcmpgtuh VRT, VRA, VRB  (Rc=0)
vcmpgtuh. VRT, VRA, VRB  (Rc=1)
```

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>22</th>
<th>582</th>
<th>31</th>
</tr>
</thead>
</table>

For each integer value \(i\) from 0 to 127, do the following.

Unsigned-integer halfword element \(i\) in VRA is compared to unsigned-integer halfword element \(i\) in VRB. Halfword element \(i\) in VRT is set to all 1s if unsigned-integer halfword element \(i\) in VRA is greater than to unsigned-integer halfword element \(i\) in VRB, and is set to all 0s otherwise.

**Special Registers Altered:**

- CR field \(6\) .............. (if \(Rc=1\))

---

Vector Compare Greater Than Unsigned Word VC-form

```
vcmpgtuw VRT, VRA, VRB  (Rc=0)
vcmpgtuw. VRT, VRA, VRB  (Rc=1)
```

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>VRT</th>
<th>11</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>21</th>
<th>22</th>
<th>646</th>
<th>31</th>
</tr>
</thead>
</table>

For each integer value \(i\) from 0 to 3, do the following.

Unsigned-integer word element \(i\) in VRA is compared to unsigned-integer word element \(i\) in VRB. Word element \(i\) in VRT is set to all 1s if unsigned-integer word element \(i\) in VRA is greater than to unsigned-integer word element \(i\) in VRB, and is set to all 0s otherwise.

**Special Registers Altered:**

- CR field \(6\) .............. (if \(Rc=1\))
5.9.3 Vector Logical Instructions

Extended mnemonics for vector logical operations

Extended mnemonics are provided that use the Vector OR and Vector NOR instructions to copy the contents of one Vector Register to another, with and without complementing. These are shown as examples with the two instructions.

**Vector Move Register**

Several vector instructions can be coded in a way such that they simply copy the contents of one Vector Register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register \( V_y \) to register \( V_x \).

\[ \text{vmr } V_x, V_y \quad (\text{equivalent to: } \text{vor } V_x, V_y, V_y) \]

**Vector Complement Register**

The Vector NOR instruction can be coded in a way such that it complements the contents of one Vector Register and places the result into another Vector Register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register \( V_y \) and places the result into register \( V_x \).

\[ \text{vnot } V_x, V_y \quad (\text{equivalent to: } \text{vnor } V_x, V_y, V_y) \]

**Vector Logical AND VX-form**

\[ \text{vand } V_{RT}, V_{RA}, V_{RB} \]

\[
\begin{array}{cccccc}
4 & 4 & 11 & 16 & 21 & 31 \\
0 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

\[ V_{RT} \leftarrow V_{RA} \land V_{RB} \]

The contents of \( V_{RA} \) are ANDed with the contents of \( V_{RB} \) and the result is placed into \( V_{RT} \).

**Special Registers Altered:** None

**Vector Logical AND with Complement VX-form**

\[ \text{vandc } V_{RT}, V_{RA}, V_{RB} \]

\[
\begin{array}{cccccc}
4 & 4 & 11 & 16 & 21 & 31 \\
0 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

\[ V_{RT} \leftarrow V_{RA} \land \neg V_{RB} \]

The contents of \( V_{RA} \) are ANDed with the complement of the contents of \( V_{RB} \) and the result is placed into \( V_{RT} \).

**Special Registers Altered:** None

**Vector Logical Equivalent VX-form**

\[ \text{veqv } V_{RT}, V_{RA}, V_{RB} \]

\[
\begin{array}{cccccc}
4 & 4 & 11 & 16 & 21 & 31 \\
0 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

\[ V_{RT} \leftarrow V_{RA} \oplus \overline{V_{RB}} \]

The contents of \( V_{RA} \) are XORed with the contents of \( V_{RB} \) and the complemented result is placed into \( V_{RT} \).

**Special Registers Altered:** None

**Vector Logical NAND VX-form**

\[ \text{vnand } V_{RT}, V_{RA}, V_{RB} \]

\[
\begin{array}{cccccc}
4 & 4 & 11 & 16 & 21 & 31 \\
0 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

\[ \text{if MSR.VEC=0 then VECTOR_UNAVAILABLE} \]

\[ V_{RT} \leftarrow \lnot (V_{RA} \land V_{RB}) \]

The contents of \( V_{RA} \) are ANDed with the contents of \( V_{RB} \) and the complemented result is placed into \( V_{RT} \).

**Special Registers Altered:** None
Vector Logical OR with Complement VX-form

\[
\text{vorc} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

\[
\begin{array}{cccccc}
4 & 0 & 6 & 11 & 16 & 21 & 1348
\end{array}
\]

\[\text{VR}[\text{VRT}] \leftarrow \text{VR}[\text{VRA}] \mid \neg \text{VR}[\text{VRB}]\]

The contents of \text{VR}[\text{VRA}] are ORed with the complement of the contents of \text{VR}[\text{VRB}] and the result is placed into \text{VR}[\text{VRT}].

Special Registers Altered:
None

Vector Logical NOR VX-form

\[
\text{vnor} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

\[
\begin{array}{cccccc}
4 & 0 & 6 & 11 & 16 & 21 & 1284
\end{array}
\]

\[\text{VR}[\text{VRT}] \leftarrow \neg (\text{VR}[\text{VRA}] \mid \text{VR}[\text{VRB}])\]

The contents of \text{VR}[\text{VRA}] are ORed with the contents of \text{VR}[\text{VRB}] and the complemented result is placed into \text{VR}[\text{VRT}].

Special Registers Altered:
None

Vector Logical OR VX-form

\[
\text{vor} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

\[
\begin{array}{cccccc}
4 & 0 & 6 & 11 & 16 & 21 & 1156
\end{array}
\]

\[\text{VR}[\text{VRT}] \leftarrow \text{VR}[\text{VRA}] \mid \text{VR}[\text{VRB}]\]

The contents of \text{VR}[\text{VRA}] are ORed with the contents of \text{VR}[\text{VRB}] and the result is placed into \text{VR}[\text{VRT}].

Special Registers Altered:
None

Vector Logical XOR VX-form

\[
\text{vxor} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

\[
\begin{array}{cccccc}
4 & 0 & 6 & 11 & 16 & 21 & 1220
\end{array}
\]

\[\text{VR}[\text{VRT}] \leftarrow \text{VR}[\text{VRA}] \oplus \text{VR}[\text{VRB}]\]

The contents of \text{VR}[\text{VRA}] are XORed with the contents of \text{VR}[\text{VRB}] and the result is placed into \text{VR}[\text{VRT}].

Special Registers Altered:
None
5.9.4 Vector Integer Rotate and Shift Instructions

### Vector Rotate Left Byte VX-form

- **Syntax:** `vrlb VRT, VRA, VRB`
- **Operation:**
  
  ```
  do i=0 to 127 by 8
  sh ← (VRB)i+5:i+7
  VRTi:i+7 ← (VRA)i:i+7 <<< sh
  end
  ```

  For each integer value `i` from 0 to 15, do the following.
  
  Byte element `i` in `VRA` is rotated left by the number of bits specified in the low-order 3 bits of the corresponding byte element `i` in `VRB`.

  The result is placed into byte element `i` in `VRT`.

  **Special Registers Altered:** None

### Vector Rotate Left Halfword VX-form

- **Syntax:** `vrlh VRT, VRA, VRB`
- **Operation:**
  
  ```
  do i=0 to 127 by 16
  sh ← (VRB)i+12:i+15
  VRTi:i+15 ← (VRA)i:i+15 <<< sh
  end
  ```

  For each integer value `i` from 0 to 7, do the following.
  
  Halfword element `i` in `VRA` is rotated left by the number of bits specified in the low-order 4 bits of the corresponding halfword element `i` in `VRB`.

  The result is placed into halfword element `i` in `VRT`.

  **Special Registers Altered:** None

### Vector Rotate Left Word VX-form

- **Syntax:** `vrlw VRT, VRA, VRB`
- **Operation:**
  
  ```
  do i=0 to 127 by 32
  sh ← (VRB)i+27:i+31
  VRTi:i+31 ← (VRA)i:i+31 <<< sh
  end
  ```

  For each integer value `i` from 0 to 3, do the following.
  
  Word element `i` in `VRA` is rotated left by the number of bits specified in the low-order 5 bits of the corresponding word element `i` in `VRB`.

  The result is placed into word element `i` in `VRT`.

  **Special Registers Altered:** None

### Vector Rotate Left Doubleword VX-form

- **Syntax:** `vrld VRT, VRA, VRB`
- **Operation:**
  
  ```
  do i=0 to 1
  sh ← VR[VRB].dword[i].bit[58:63]
  VR[VRT].dword[i] ← VR[VRA].dword[i] <<< sh
  end
  ```

  For each integer value `i` from 0 to 1, do the following.
  
  The contents of doubleword element `i` of `VR(VRA)` are rotated left by the number of bits specified in bits 58:63 of doubleword element `i` of `VR(VRB)`.

  The result is placed into doubleword element `i` of `VR(VRT)`.

  **Special Registers Altered:** None
Vector Shift Left Byte VX-form

**vslb** VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>260</th>
<th>31</th>
</tr>
</thead>
</table>

For each integer value i from 0 to 15, do the following.
Byte element i in VRA is shifted left by the number of bits specified in the low-order 3 bits of byte element i in VRB.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Shift Left Halfword VX-form

**vslh** VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>324</th>
<th>31</th>
</tr>
</thead>
</table>

For each integer value i from 0 to 7, do the following.
Halfword element i in VRA is shifted left by the number of bits specified in the low-order 4 bits of halfword element i in VRB.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Shift Left Word VX-form

**vslw** VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>388</th>
<th>31</th>
</tr>
</thead>
</table>

do i=0 to 127 by 8
sh ← (VRB)i+5:i+7
VRTi:i+7 ← (VRA)i:i+7 << sh
end

For each integer value i from 0 to 15, do the following.
Word element i in VRA is shifted left by the number of bits specified in the low-order 5 bits of word element i in VRB.

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into word element i of VRT.

Special Registers Altered:
None

Vector Shift Left Doubleword VX-form

**vsld** VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1476</th>
<th>31</th>
</tr>
</thead>
</table>

do i = 0 to 1
sh ← VR[VRB].dword[i].bit[58:63]
VRT.[VRB].dword[i] ← VR[VRA].dword[i] << sh
end

For each integer value i from 0 to 1, do the following.
The contents of doubleword element i of VR[VRA] are shifted left by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB].

- Bits shifted out of bit 0 are lost.
- Zeros are supplied to the vacated bits on the right.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None
### Vector Shift Right Byte VX-form

**vsrb**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>516</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

---

VX-form

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 8 \\
\text{sh } \leftarrow (\text{VRB})_8:i+7 \\
\text{VRT}_i:i+7 \leftarrow (\text{VRA})_8:i+7 >>_8 \text{ sh} \\
\text{end}
\]

For each integer value \( i \) from 0 to 15, do the following.

Byte element \( i \) in VRA is shifted right by the number of bits specified in the low-order 3 bits of byte element \( i \) in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into byte element \( i \) of VRT.

**Special Registers Altered:**

None

### Vector Shift Right Word VX-form

**vsrw**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>580</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

---

VX-form

\[
\text{do } i = 0 \text{ to } 127 \text{ by } 16 \\
\text{sh } \leftarrow (\text{VRB})_{16}:i+15 \\
\text{VRT}_i:i+15 \leftarrow (\text{VRA})_{16}:i+15 >>_{16} \text{ sh} \\
\text{end}
\]

For each integer value \( i \) from 0 to 7, do the following.

Halfword element \( i \) in VRA is shifted right by the number of bits specified in the low-order 4 bits of halfword element \( i \) in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into halfword element \( i \) of VRT.

**Special Registers Altered:**

None

### Vector Shift Right Doubleword VX-form

**vsrd**

<table>
<thead>
<tr>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1732</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

---

VX-form

\[
\text{do } i = 0 \text{ to } 1 \\
\text{sh } \leftarrow \text{VRB}_i >>_{64} \text{ sh} \\
\text{VR}_i \leftarrow \text{VRB}_i >>_{64} \text{ sh} \\
\text{end}
\]

For each integer value \( i \) from 0 to 1, do the following.

The contents of doubleword element \( i \) of VR\[VRA\] are shifted right by the number of bits specified in bits 58:63 of doubleword element \( i \) of VRT[VRB]. Zeros are supplied to the vacated bits on the left.

The result is placed into doubleword element \( i \) of VRT[VRT].

**Special Registers Altered:**

None
Vector Shift Right Algebraic Byte VX-form
vsrab VRT,VRA,VRB

<table>
<thead>
<tr>
<th>do i = 0 to 127 by 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh ← (VRB)i+5:i+7</td>
</tr>
<tr>
<td>VRTi:i+7 ← (VRA)i:i+7 &gt;&gt;si sh</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 15, do the following. Byte element i in VRA is shifted right by the number of bits specified in the low-order 3 bits of the corresponding byte element i in VRB. Bits shifted out of bit 7 of the byte element are lost. Bit 0 of the byte element is replicated to fill the vacated bits on the left. The result is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Shift Right Algebraic Halfword VX-form
vsrah VRT,VRA,VRB

<table>
<thead>
<tr>
<th>do i = 0 to 127 by 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh ← (VRB)i+12:i+15</td>
</tr>
<tr>
<td>VRTi:i+15 ← (VRA)i:i+15 &gt;&gt;si sh</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 7, do the following. Halfword element i in VRA is shifted right by the number of bits specified in the low-order 4 bits of the corresponding halfword element i in VRB. Bits shifted out of bit 15 of the halfword are lost. Bit 0 of the halfword is replicated to fill the vacated bits on the left. The result is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Shift Right Algebraic Word VX-form
vsraw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>do i = 0 to 127 by 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh ← (VRB)i+27:i+31</td>
</tr>
<tr>
<td>VRTi+31 ← (VRA)i:i+31 &gt;&gt;si sh</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following. Word element i in VRA is shifted right by the number of bits specified in the low-order 5 bits of the corresponding word element i in VRB. Bits shifted out of bit 31 of the word are lost. Bit 0 of the word is replicated to fill the vacated bits on the left. The result is placed into word element i of VRT.

Special Registers Altered:
None

Vector Shift Right Algebraic Doubleword VX-form
vsrad VRT,VRA,VRB

<table>
<thead>
<tr>
<th>do i = 0 to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh ← VR[VRB].dword[i].bit[58:63]</td>
</tr>
<tr>
<td>VR[VRT].dword[i] ← VR[VRA].dword[i] &gt;&gt;si sh</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 1, do the following. The contents of doubleword element i of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB]. Bit 0 of doubleword element i of VR[VRA] is replicated to fill the vacated bits on the left. The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None
## 5.10 Vector Floating-Point Instruction Set

### 5.10.1 Vector Floating-Point Arithmetic Instructions

**Vector Add Single-Precision VX-form**

```plaintext
vaddfp VRT, VRA, VRB
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is added to single-precision floating-point element i in VRB. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

**Special Registers Altered:**
None

**Vector Subtract Single-Precision VX-form**

```plaintext
vsubfp VRT, VRA, VRB
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is subtracted from single-precision floating-point element i in VRA. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

**Special Registers Altered:**
None
**Vector Multiply-Add Single-Precision**

VA-form

\[
\text{vmaddfp } \quad \text{VRT, VRA, VRC, VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>VRC</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
prod \leftarrow (\text{VRA}[i:i+31] \times \text{fp}(\text{VRC}[i:i+31])
VRT[i:i+31] \leftarrow \text{RoundToNearSP(} \text{prod +fp(}\text{VRB}[i:i+31]) \text{)}
end

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is multiplied by single-precision floating-point
element i in VRC. Single-precision floating-point
element i in VRB is added to the infinitely-precise
product. The intermediate result is rounded to the
nearest single-precision floating-point number and
placed into word element i of VRT.

**Special Registers Altered:**
None

**Programming Note**
To use a multiply-add to perform an IEEE or Java
compliant multiply, the addend must be -0.0. This
is necessary to insure that the sign of a zero result
will be correct when the product is -0.0 (+0.0 + -0.0
\geq +0.0, and -0.0 + -0.0 \geq -0.0). When the sign of a
resulting 0.0 is not important, then +0.0 can be
used as an addend which may, in some cases,
avoid the need for a second register to hold a -0.0
in addition to the integer/floating-point +0.0 that
may already be available.

**Vector Negative Multiply-Subtract**

Single-Precision VA-form

\[
\text{vnmsubfp } \quad \text{VRT, VRA, VRC, VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>26</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VRT</td>
<td>VRA</td>
<td>VRB</td>
<td>VRC</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
prod_{i,inf} \leftarrow (\text{VRA}[i:i+31] \times \text{fp}(\text{VRC}[i:i+31])
VRT[i:i+31] \leftarrow \text{-RoundToNearSP(} \text{prod_{i,inf +fp(}\text{VRB}[i:i+31])} \text{)}
end

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is multiplied by single-precision floating-point
element i in VRC. Single-precision floating-point
element i in VRB is subtracted from the
infinitely-precise product. The intermediate result
is rounded to the nearest single-precision floating-point
number, then negated and placed
into word element i of VRT.

**Special Registers Altered:**
None
5.10.2 Vector Floating-Point Maximum and Minimum Instructions

**Vector Maximum Single-Precision VX-form**

\[ \text{vmaxfp} \quad \text{VRT, VRA, VRB} \]

\[
\begin{array}{c|cccc|c}
4 & 6 & 11 & 16 & 21 & 1034 \\
\hline
0 & 10 & 10 & 10 & 10 & 10 \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.

Single-precision floating-point element \( i \) in VRA is compared to single-precision floating-point element \( i \) in VRB. The larger of the two values is placed into word element \( i \) of VRT.

The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

**Special Registers Altered:**

None

---

**Vector Minimum Single-Precision VX-form**

\[ \text{vminfp} \quad \text{VRT, VRA, VRB} \]

\[
\begin{array}{c|cccc|c}
4 & 6 & 11 & 16 & 21 & 1098 \\
\hline
0 & 10 & 10 & 10 & 10 & 10 \\
\end{array}
\]

For each integer value \( i \) from 0 to 3, do the following.

Single-precision floating-point element \( i \) in VRA is compared to single-precision floating-point element \( i \) in VRB. The smaller of the two values is placed into word element \( i \) of VRT.

The minimum of +0 and -0 is -0. The minimum of any value and a NaN is a QNaN.

**Special Registers Altered:**

None
5.10.3 Vector Floating-Point Rounding and Conversion Instructions

See Appendix C, “Vector RTL Functions [Category: Vector]” on page 701, for RTL function descriptions.

Vector Convert To Signed Fixed-Point Word Saturate VX-form

\[ \text{vctsxs} \quad \text{VRT,VRB,UIM} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

\[
\text{do } i=0 \text{ to 127 by 32 } \\
\text{VRT}_{i:i+31} \leftarrow \text{ConvertSPtoSXW}saturate((\text{VRB})_{i:i+31}, \text{UIM}) \]

For each integer value \(i\) from 0 to 3, do the following.

- Single-precision floating-point word element \(i\) in VRB is multiplied by \(2^{\text{UIM}}\). The product is converted to a 32-bit signed fixed-point integer using the rounding mode Round toward Zero.
  - If the intermediate result is greater than \(2^{31}-1\) the result saturates to \(2^{31}-1\).
  - If the intermediate result is less than \(-2^{31}\) the result saturates to \(-2^{31}\).

The result is placed into word element \(i\) of VRT.

Special Registers Altered: SAT

Extended Mnemonics:
Example of an extended mnemonics for Vector Convert To Signed Fixed-Point Word Saturate:

\[
\text{Extended: } \quad \text{vcfpsxws} \quad \text{VRT,VRB,UIM} \\
\text{Equivalent to: } \quad \text{vctsxs} \quad \text{VRT,VRB,UIM}
\]

Vector Convert To Unsigned Fixed-Point Word Saturate VX-form

\[ \text{vctuxs} \quad \text{VRT,VRB,UIM} \]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
</table>

\[
\text{do } i=0 \text{ to 127 by 32 } \\
\text{VRT}_{i:i+31} \leftarrow \text{ConvertSPtoUXW}saturate((\text{VRB})_{i:i+31}, \text{UIM}) \]

For each integer value \(i\) from 0 to 3, do the following.

- Single-precision floating-point word element \(i\) in VRB is multiplied by \(2^{\text{UIM}}\). The product is converted to a 32-bit unsigned fixed-point integer using the rounding mode Round toward Zero.
  - If the intermediate result is greater than \(2^{32}-1\) the result saturates to \(2^{32}-1\).

The result is placed into word element \(i\) of VRT.

Special Registers Altered: SAT

Extended Mnemonics:
Example of an extended mnemonics for Vector Convert to Unsigned Fixed-Point Word Saturate:

\[
\text{Extended: } \quad \text{vcfpuwsx} \quad \text{VRT,VRB,UIM} \\
\text{Equivalent to: } \quad \text{vctuxs} \quad \text{VRT,VRB,UIM}
\]
### Vector Convert From Signed Fixed-Point Word VX-form

**vcfsx**

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>UIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ \text{do } i=0 \text{ to } 127 \text{ by } 32 \]

\[ \text{VRT}_{i:i+31} \leftarrow \text{ConvertSXWtoSP}( (\text{VRB})_{i:i+31} ) \div \text{fp } 2^{\text{UIM}} \]

\[ \text{end} \]

For each integer value \( i \) from 0 to 3, do the following. Signed fixed-point word element \( i \) in \text{VRB} is converted to the nearest single-precision floating-point value. Each result is divided by \( 2^{\text{UIM}} \) and placed into word element \( i \) of \text{VRT}.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Examples of extended mnemonics for **Vector Convert from Signed Fixed-Point Word**

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcxsxwp</td>
<td>vcfsx VRT,VRB,UIM</td>
</tr>
</tbody>
</table>

### Vector Convert From Unsigned Fixed-Point Word VX-form

**vcfux**

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>UIM</th>
<th>VRB</th>
<th>UIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ \text{do } i=0 \text{ to } 127 \text{ by } 32 \]

\[ \text{VRT}_{i:i+31} \leftarrow \text{ConvertUXWtoSP}( (\text{VRB})_{i:i+31} ) \div \text{fp } 2^{\text{UIM}} \]

\[ \text{end} \]

For each integer value \( i \) from 0 to 3, do the following. Unsigned fixed-point word element \( i \) in \text{VRB} is converted to the nearest single-precision floating-point value. The result is divided by \( 2^{\text{UIM}} \) and placed into word element \( i \) of \text{VRT}.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Examples of extended mnemonics for **Vector Convert from Unsigned Fixed-Point Word**

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcuxwp</td>
<td>vcfux VRT,VRB,UIM</td>
</tr>
</tbody>
</table>
Vector Round to Single-Precision Integer
toward -Infinity VX-form

vrfin VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>714</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
VRT0:31 ← RoundToSPIntFloor{(VRB)0:31}
end

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward -Infinity.
The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Programming Note

The Vector Convert To Fixed-Point Word instructions support only the rounding mode Round toward Zero. A floating-point number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate Vector Round to Floating-Point Integer instruction before the Vector Convert To Fixed-Point Word instruction.

Programming Note

The fixed-point integers used by the Vector Convert instructions can be interpreted as consisting of 32-UIM integer bits followed by UIM fraction bits.

Vector Round to Single-Precision Integer
Nearest VX-form

vrfin VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>522</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

doi=0 to 127 by 32
VRT0:31 ← RoundToSPIntNear{(VRB)0:31}
end

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round to Nearest.
The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Vector Round to Single-Precision Integer
toward +Infinity VX-form

vrfip VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>650</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

doi=0 to 127 by 32
VRT0:31 ← RoundToSPIntCeil{(VRB)0:31}
end

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity.
The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Programming Note

The Vector Convert To Fixed-Point Word instructions support only the rounding mode Round toward Zero. A floating-point number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate Vector Round to Floating-Point Integer instruction before the Vector Convert To Fixed-Point Word instruction.
**Vector Round to Single-Precision Integer toward Zero VX-form**

vr.gz \ VRT,VRB

\[
\begin{array}{cccccc}
0 & 4 & 6 & 31 & 21 & 586 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\end{array}
\]

do i=0 to 127 by 32

VRT0:31 \ RoundToSPIntTrunc( (VRB)0:31 | 
end

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward Zero.

The result is placed into the corresponding word element i of VRT.

**Special Registers Altered:**

None
5.10.4 Vector Floating-Point Compare Instructions

The Vector Floating-Point Compare instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the Vector Compare Bounds Floating-Point instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the Vector Integer Compare instructions; see Section 5.9.2.

The Vector Compare Bounds Floating-Point instruction sets the target Vector Register, and CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be “within the bounds” specified by a single-precision floating-point value y if -y ≤ x ≤ y.

---

**Vector Compare Bounds Single-Precision VC-form**

<table>
<thead>
<tr>
<th>vcmpbfp</th>
<th>VRT, VRA, VRB (Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcmpbf</td>
<td>VRT, VRA, VRB (Rc=1)</td>
</tr>
</tbody>
</table>

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRA is compared to single-precision floating-point word element i in VRB. A 2-bit value is formed that indicates whether the element in VRA is within the bounds specified by the element in VRB, as follows.

- Bit 0 of the 2-bit value is set to 0 if the element in VRA is less than or equal to the element in VRB, and is set to 1 otherwise.
- Bit 1 of the 2-bit value is set to 0 if the element in VRA is greater than or equal to the negation of the element in VRB, and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element i of VRT and the remaining bits of element i are set to 0.

If Rc=1, CR field 6 is set as follows.

**Bit Description**

| 0 | Set to 0 |
| 1 | Set to 0 |

**Programming Note**

Each single-precision floating-point word element in VRB should be non-negative; if it is negative, the corresponding element in VRA will necessarily be out of bounds.

One exception to this is when the value of an element in VRB is -0.0 and the value of the corresponding element in VRA is either +0.0 or -0.0. +0.0 and -0.0 compare equal to -0.0.
### Vector Compare Equal To
#### Single-Precision VC-form

**vcmeqfp**  
\( VRT, VRA, VRB \)  
(Rc=0)

**vcmeqfp**  
\( VRT, VRA, VRB \)  
(Rc=1)

```
do i=0 to 127 by 32
   VRT(I:i+31) = (VRA(I:i+31) == fp (VRB(I:i+31))) ? 321 : 320
end
```

**Special Registers Altered:**

- **CR field 6** (if Rc=1)

---

### Vector Compare Greater Than or Equal To
#### Single-Precision VC-form

**vcmpgefp**  
\( VRT, VRA, VRB \)  
(Rc=0)

**vcmpgefp**  
\( VRT, VRA, VRB \)  
(Rc=1)

```
do i=0 to 127 by 32
   VRT(I:i+31) = (VRA(I:i+31) \geq fp (VRB(I:i+31))) ? 321 : 320
end
```

**Special Registers Altered:**

- **CR field 6** (if Rc=1)
Vector Compare Greater Than
Single-Precision VC-form

vcmpgtfp VRT, VRA, VRB \ (Rc=0) 
vcmpgtfp, VRT, VRA, VRB \ (Rc=1)

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
</table>

do i=0 to 127 by 32 
  VRT_i:i+31 \ = \ (VRA_i:i+31 >fp (VRB)_i:i+31) ? 321 : 320 
end 
if Rc=1 then do 
t \ = \ (VRT_128:1) 
f \ = \ (VRT_128:0) 
CR6 \ leftarrow t \ || \ 0b0 \ || f \ || \ 0b0 
end

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is greater than single-precision floating-point element i in VRB, and is set to all 0s otherwise.

If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 0s, indicating “not greater than”.

Special Registers Altered:
CR field 6 \ ......................... \ (if Rc=1)
5.10.5 Vector Floating-Point Estimate Instructions

**Vector 2 Raised to the Exponent Estimate Floating-Point VX-form**

```
vexptefp VRT,VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>394</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>
```

For each integer value \( i \) from 0 to 3, do the following.

The single-precision floating-point estimate of \( 2 \) raised to the power of single-precision floating-point element \( i \) in VRB is placed into word element \( i \) of VRT.

Let \( x \) be any single-precision floating-point input value. Unless \( x < -146 \) or the single-precision floating-point result of computing \( 2 \) raised to the power \( x \) would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16. The most significant 12 bits of the estimate’s significand are monotonic. An integral input value returns an integral value when the result is representable.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Infinity</td>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
</tr>
<tr>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**

None

**Vector Log Base 2 Estimate Floating-Point VX-form**

```
vlogefp VRT,VRB
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>458</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>
```

For each integer value \( i \) from 0 to 3, do the following.

The single-precision floating-point estimate of the base 2 logarithm of single-precision floating-point element \( i \) in VRB is placed into the corresponding word element of VRT.

Let \( x \) be any single-precision floating-point input value. Unless \( |x-1| \leq 0.125 \) or the single-precision floating-point result of computing the base 2 logarithm of \( x \) would be an infinity or a QNaN, the estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than \( 2^{-5} \). Under the same conditions, the estimate has a relative error in precision no greater than one part in 8.

The most significant 12 bits of the estimate’s significand are monotonic. The estimate is exact if \( x=2^y \), where \( y \) is an integer between -149 and +127 inclusive. Otherwise the value placed into the element of register VRT may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>QNaN</td>
</tr>
<tr>
<td>-0</td>
<td>- Infinity</td>
</tr>
<tr>
<td>+0</td>
<td>- Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**

None
**Vector Reciprocal Estimate**

**Single-Precision VX-form**

\[ \text{vrefp } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>i</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 127, do the following.

The single-precision floating-point estimate of the reciprocal of single-precision floating-point element \( i \) in \( VRB \) is placed into word element \( i \) of \( VRT \).

Unless the single-precision floating-point result of computing the reciprocal of a value would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in \( 4096 \).

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Infinity</td>
<td>-0</td>
</tr>
<tr>
<td>- 0</td>
<td>- Infinity</td>
</tr>
<tr>
<td>+0</td>
<td>+ Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+0</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:

None

---

**Vector Reciprocal Square Root Estimate**

**Single-Precision VX-form**

\[ \text{vrsqrtefp } \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th>i</th>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each integer value \( i \) from 0 to 127, do the following.

The single-precision floating-point estimate of the reciprocal of the square root of single-precision floating-point element \( i \) in \( VRB \) is placed into word element \( i \) of \( VRT \).

Let \( x \) be any single-precision floating-point value. Unless the single-precision floating-point result of computing the reciprocal of the square root of \( x \) would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in \( 4096 \).

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>QNaN</td>
</tr>
<tr>
<td>- 0</td>
<td>- Infinity</td>
</tr>
<tr>
<td>+0</td>
<td>+ Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+0</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

Special Registers Altered:

None
5.11 Vector Exclusive-OR-based Instructions

5.11.1 Vector AES Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 197 Advanced Encryption Standard for encryption and decryption.

**Vector AES Cipher VX-form**

```
vcipher VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1288</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

State ← VR[VRA]  
RoundKey ← VR[VRB]  
vtemp1 ← SubBytes(State)  
vtemp2 ← ShiftRows(vtemp1)  
vtemp3 ← MixColumns(vtemp2)  
VR[VRT] ← vtemp3 ^ RoundKey

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

One round of an AES cipher operation is performed on the intermediate State array, sequentially applying the transforms, SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the new intermediate state of the cipher operation.

**Special Registers Altered:**  
None

**Vector AES Cipher Last VX-form**

```
vcipherlast VRT,VRA,VRB
```

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1289</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

State ← VR[VRA]  
RoundKey ← VR[VRB]  
vtemp1 ← SubBytes(State)  
vtemp2 ← ShiftRows(vtemp1)  
VR[VRT] ← vtemp2 ^ RoundKey

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

The final round in an AES cipher operation is performed on the intermediate State array, sequentially applying the transforms, SubBytes(), ShiftRows(), AddRoundKey(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the final state of the cipher operation.

**Special Registers Altered:**  
None
**Vector AES Inverse Cipher VX-form**  
[Category: Vector.Crypto]

\[
\text{vncipher} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1352</th>
</tr>
</thead>
</table>

State $\leftarrow \text{VR}[\text{VRA}]
\text{RoundKey} \leftarrow \text{VR}[\text{VRB}]
\text{vtemp1} \leftarrow \text{InvShiftRows}(\text{State})
\text{vtemp2} \leftarrow \text{InvSubBytes}(\text{vtemp1})
\text{vtemp3} \leftarrow \text{InvMixColumns}(\text{vtemp2})
\text{VR}[\text{VRT}] \leftarrow \text{vtemp3} \oplus \text{RoundKey}

Let State be the contents of VR[VRA], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

One round of an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), and InvMixColumns(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the new intermediate state of the inverse cipher operation.

Special Registers Altered:
None

**Vector AES Inverse Cipher Last VX-form**  
[Category: Vector.Crypto]

\[
\text{vncipherlast} \quad \text{VRT}, \text{VRA}, \text{VRB}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1353</th>
</tr>
</thead>
</table>

State $\leftarrow \text{VR}[\text{VRA}]
\text{RoundKey} \leftarrow \text{VR}[\text{VRB}]
\text{vtemp1} \leftarrow \text{InvShiftRows}(\text{State})
\text{vtemp2} \leftarrow \text{InvSubBytes}(\text{vtemp1})
\text{VR}[\text{VRT}] \leftarrow \text{vtemp2} \oplus \text{RoundKey}

Let State be the contents of VR[VRA], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

The final round in an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), and AddRoundKey(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the final state of the inverse cipher operation.

Special Registers Altered:
None

**Vector AES SubBytes VX-form**  
[Category: Vector.Crypto]

\[
\text{vsbox} \quad \text{VRT}, \text{VRA}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1480</th>
</tr>
</thead>
</table>

State $\leftarrow \text{VR}[\text{VRA}]
\text{VR}[\text{VRT}] \leftarrow \text{SubBytes}(\text{State})

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

The result of applying the transform, SubBytes() on State, as defined in FIPS-197, is placed into VR[VRT].

Special Registers Altered:
None
5.11.2 Vector SHA-256 and SHA-512 Sigma Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 180-3 Secure Hash Standard.

**Vector SHA-512 Sigma Doubleword VX-form**
[Category:Vector.Crypto]

<table>
<thead>
<tr>
<th>vhsasigmad</th>
<th>VRT,VRA,ST,SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 6 11 16 17 21 31</td>
<td>1730</td>
</tr>
</tbody>
</table>

Do i = 0 to 1

src ← VR[VRA].doubleword[i]

if ST=0 & SIX.bit[2*i]=0 then // SHA-512 σ0 function
   VR[VRT].dword[i] ← (src >>> 1) *
   | src >>> 8 |
   | src >> 7 |
if ST=0 & SIX.bit[2*i]=1 then // SHA-512 σ0 function
   VR[VRT].dword[i] ← (src >>> 1) *
   | src >>> 8 |
   | src >> 7 |
if ST=1 & SIX.bit[2*i]=0 then // SHA-512 Σ0 function
   VR[VRT].dword[i] ← (src >>> 19) *
   | src >>> 6 |
   | src >> 10 |
if ST=1 & SIX.bit[2*i]=1 then // SHA-512 Σ0 function
   VR[VRT].dword[i] ← (src >>> 61) *
   | src >> 6 |
   | src >>> 3 |

For each integer value i from 0 to 1, do the following.

When ST=0 and bit 2×i of SIX is 0, a SHA-512 σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=0 and bit 2×i of SIX is 1, a SHA-512 σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 0, a SHA-512 Σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 1, a SHA-512 Σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

Bits 1 and 3 of SIX are reserved.

**Special Registers Altered:**
None

**Vector SHA-256 Sigma Word VX-form**
[Category:Vector.Crypto]

<table>
<thead>
<tr>
<th>vhsasigmaw</th>
<th>VRT,VRA,ST,SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 6 11 16 17 21 31</td>
<td>1666</td>
</tr>
</tbody>
</table>

Do i = 0 to 3

src ← VR[VRA].word[i]

if ST=0 & SIX.bit[i]=0 then // SHA-256 σ0 function
   VR[VRT].word[i] ← (src >>> 7) *
   | src >>> 18 |
   | src >> 6 |
if ST=0 & SIX.bit[i]=1 then // SHA-256 σ0 function
   VR[VRT].word[i] ← (src >>> 17) *
   | src >>> 19 |
   | src >> 10 |
if ST=1 & SIX.bit[i]=0 then // SHA-256 Σ0 function
   VR[VRT].word[i] ← (src >>> 2) *
   | src >>> 13 |
   | src >> 22 |
if ST=1 & SIX.bit[i]=1 then // SHA-256 Σ0 function
   VR[VRT].word[i] ← (src >>> 6) *
   | src >>> 11 |
   | src >> 25 |

For each integer value i from 0 to 3, do the following.

When ST=0 and bit i of SIX is 0, a SHA-256 σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=0 and bit i of SIX is 1, a SHA-256 σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=1 and bit i of SIX is 0, a SHA-256 Σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=1 and bit i of SIX is 1, a SHA-256 Σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

**Special Registers Altered:**
None
5.11.3 Vector Binary Polynomial Multiplication Instructions

This section describes a set of binary polynomial multiply-sum instructions. Corresponding elements are multiplied and the exclusive-OR of each even-odd pair of products summ, useful for a variety of finite field arithmetic operations.

**Vector Polynomial Multiply-Sum Byte VX-form**

```
vpmsumb VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1032</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
prod[i].bit[0:14] ← 0
srcA ← VR[VRA].byte[i]
srcB ← VR[VRB].byte[i]
do j = 0 to 7
   do k = 0 to j
      gbit ← srcA.bit[k] & srcB.bit[j-k]
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
   end
end
do j = 8 to 14
   do k = j-7 to 7
      gbit ← (srcA.bit[k] & srcB.bit[j-k])
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
   end
end
do i = 0 to 7
VR[VRT].hword[i] ← 0b0 ^ (prod[2×i] ^ prod[2×i+1])
end
```

For each integer value i from 0 to 15, do the following.
Let prod[i] be the 15-bit result of a binary polynomial multiplication of the contents of byte element i of VR[VRA] and the contents of byte element i of VR[VRB].

For each integer value i from 0 to 7, do the following.
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:15 of halfword element i of VR[VRT]. Bit 0 of halfword element i of VR[VRT] is set to 0.

Special Registers Altered:
None

**Vector Polynomial Multiply-Sum Doubleword VX-form**

```
vpmsumd VRT,VRA,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1224</th>
</tr>
</thead>
</table>
```

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
prod[i].bit[0:126] ← 0
srcA ← VR[VRA].doubleword[i]
srcB ← VR[VRB].doubleword[i]
do j = 0 to 63
   do k = 0 to j
      gbit ← srcA.bit[k] & srcB.bit[j-k]
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
   end
end
do j = 64 to 126
   do k = j-63 to 63
      gbit ← (srcA.bit[k] & srcB.bit[j-k])
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
   end
end
do i = 0 to 7
VR[VRT] ← 0b0 ^ (prod[0] ^ prod[1])
end
```

Let prod[0] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 0 of VR[VRA] and the contents of doubleword element 0 of VR[VRB].

Let prod[1] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 1 of VR[VRA] and the contents of doubleword element 1 of VR[VRB].

The exclusive-OR of prod[0] and prod[1] is placed in bits 1:127 of VR[VRT]. Bit 0 of VR[VRT] is set to 0.

Special Registers Altered:
None
**Vector Polynomial Multiply-Sum Halfword VX-form**

```
vpmshmh VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1096</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

```do i = 0 to 7
  prod.bit[0:30] ← 0
  srcA ← VR[VRA].halfword[i]
  srcB ← VR[VRB].halfword[i]
  do j = 0 to 15
    do k = 0 to j
      gbit ← srcA.bit[k] & srcB.bit[j-k]
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  do j = 16 to 30
    do k = j-15 to 15
      gbit ← (srcA.bit[k] & srcB.bit[j-k])
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  do j = 31 to 62
    do k = j-31 to 15
      gbit ← (srcA.bit[k] & srcB.bit[j-k])
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  VR[VRT].word[0] ← 0b0 || prod[0] ^ prod[1]
```

For each integer value \( i \) from 0 to 7, do the following.

Let \( \text{prod}[i] \) be the 31-bit result of a binary polynomial multiplication of the contents of halfword element \( i \) of \( \text{VR}[\text{VRA}] \) and the contents of halfword element \( i \) of \( \text{VR}[\text{VRB}] \).

For each integer value \( i \) from 0 to 3, do the following.

The exclusive-OR of \( \text{prod}[2i] \) and \( \text{prod}[2i+1] \) is placed in bits 1:31 of word element \( i \) of \( \text{VR}[\text{VRT}] \). Bit 0 of word element \( i \) of \( \text{VR}[\text{VRT}] \) is set to 0.

**Special Registers Altered:**

None

---

**Vector Polynomial Multiply-Sum Word VX-form**

```
vpmshmw VRT,VRA,VRB

<table>
<thead>
<tr>
<th>i</th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1160</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

```do i = 0 to 3
  prod[i].bit[0:62] ← 0
  srcA ← VR[VRA].word[i]
  srcB ← VR[VRB].word[i]
  do j = 0 to 31
    do k = 0 to j
      gbit ← srcA.bit[k] & srcB.bit[j-k]
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  do j = 32 to 62
    do k = j-31 to 31
      gbit ← (srcA.bit[k] & srcB.bit[j-k])
      prod[i].bit[j] ← prod[i].bit[j] ^ gbit
    end
  end
  VR[VRT].dword[0] ← 0b0 || (prod[0] ^ prod[1])
```

For each integer value \( i \) from 0 to 3, do the following.

Let \( \text{prod}[i] \) be the 63-bit result of a binary polynomial multiplication of the contents of word element \( i \) of \( \text{VR}[\text{VRA}] \) and the contents of word element \( i \) of \( \text{VR}[\text{VRB}] \).

For each integer value \( i \) from 0 to 1, do the following.

The exclusive-OR of \( \text{prod}[2i] \) and \( \text{prod}[2i+1] \) is placed in bits 1:63 of doubleword element \( i \) of \( \text{VR}[\text{VRT}] \). Bit 0 of doubleword element \( i \) of \( \text{VR}[\text{VRT}] \) is set to 0.

**Special Registers Altered:**

None
5.11.4 Vector Permute and Exclusive-OR Instruction

**Vector Permute and Exclusive-OR**  
**VA-form**  
[Category: Vector, RAID]

\[
\text{vpermxor} \quad \text{VRT}, \text{VRA}, \text{VRB}, \text{VRC}
\]

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>VRB</th>
<th>VRC</th>
<th>4S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
do & \ i = 0 \ \text{to} \ 15 \\
\text{indexA} & \leftarrow \text{VR}^{[\text{VRC}].\text{byte}[i].\text{bit}[0:3]} \\
\text{indexB} & \leftarrow \text{VR}^{[\text{VRC}].\text{byte}[i].\text{bit}[4:7]} \\
\text{src1} & \leftarrow \text{VR}^{[\text{VRA}].\text{byte}[\text{indexA}]} \\
\text{src2} & \leftarrow \text{VR}^{[\text{VRB}].\text{byte}[\text{indexB}]} \\
\text{VSR}^{[\text{VRT}].\text{byte}[i]} & \leftarrow \text{src1} \ ^{\land} \text{src2}
\end{align*}
\]

For each integer value \( i \) from 0 to 15, do the following.

Let \( \text{indexA} \) be the contents of bits 0:3 of byte element \( i \) of \( \text{VR}^{[\text{VRC}]} \).

Let \( \text{indexB} \) be the contents of bits 4:7 of byte element \( i \) of \( \text{VR}^{[\text{VRC}]} \).

The exclusive OR of the contents of byte element \( \text{indexA} \) of \( \text{VR}^{[\text{VRA}]} \) and the contents of byte element \( \text{indexB} \) of \( \text{VR}^{[\text{VRB}]} \) is placed into byte element \( i \) of \( \text{VR}^{[\text{VRT}]} \).

**Special Registers Altered:**

None
5.12 Vector Gather Instruction

**Vector Gather Bits by Bytes by Doubleword VX-form**

\[ \text{vgbbd} \quad \text{VRT,VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th></th>
<th>VRB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>1292</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 1 \quad \text{do } j = 0 \text{ to } 7 \\
&\quad \text{do } k = 0 \text{ to } 7 \\
&\quad \quad b \leftarrow \text{VSR[VRB].dword[i].byte[k].bit[j]} \\
&\quad \quad \text{VSR[VRT].dword[i].byte[j].bit[k]} \leftarrow b \\
&\quad \quad \text{end} \\
&\quad \text{end} \\
&\quad \text{end}
\end{align*}
\]

Let \( \text{src} \) be the contents of \( \text{VR[VRB]} \), composed of two doubleword elements numbered 0 and 1.

Let each doubleword element be composed of eight bytes numbered 0 through 7.

An 8-bit × 8-bit bit-matrix transpose is performed on the contents of each doubleword element of \( \text{VR[VRB]} \) (see Figure 70).

For each integer value \( i \) from 0 to 1, do the following,

The contents of bit 0 of each byte of doubleword element \( i \) of \( \text{VR[VRB]} \) are concatenated and placed into byte 0 of doubleword element \( i \) of \( \text{VR[VRT]} \).

The contents of bit 1 of each byte of doubleword element \( i \) of \( \text{VR[VRB]} \) are concatenated and placed into byte 2 of doubleword element \( i \) of \( \text{VR[VRT]} \).

The contents of bit 3 of each byte of doubleword element \( i \) of \( \text{VR[VRB]} \) are concatenated and placed into byte 4 of doubleword element \( i \) of \( \text{VR[VRT]} \).

The contents of bit 5 of each byte of doubleword element \( i \) of \( \text{VR[VRB]} \) are concatenated and placed into byte 6 of doubleword element \( i \) of \( \text{VR[VRT]} \).

The contents of bit 7 of each byte of doubleword element \( i \) of \( \text{VR[VRB]} \) are concatenated and placed into byte 7 of doubleword element \( i \) of \( \text{VR[VRT]} \).

**Special Registers Altered:**

None

---

**Figure 70. Vector Gather Bits by Bytes by Doubleword**
### 5.13 Vector Count Leading Zeros Instructions

#### Vector Count Leading Zeros Byte VX-form

**Vclzb**  
VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1794</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```assembly
do i = 0 to 15
n ← 0
do while n < 8
  if VR[VRB].byte[i].bit[n] = 0b1 then leave
  n ← n + 1
end
VSR[VRT].byte[i] ← n
end
```

For each integer value i from 0 to 15, do the following.
A count of the number of consecutive zero bits starting at bit 0 of byte element i of VR[VRB] is placed into byte element i of VR[VRT]. This number ranges from 0 to 8, inclusive.

**Special Registers Altered:**  
None

#### Vector Count Leading Zeros Halfword VX-form

**Vclzh**  
VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1858</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```assembly
do i = 0 to 7
n ← 0
do while n < 16
  if VR[VRB].hword[i].bit[n] = 0b1 then leave
  n ← n + 1
end
VSR[VRT].hword[i] ← n
end
```

For each integer value i from 0 to 7, do the following.
A count of the number of consecutive zero bits starting at bit 0 of halfword element i of VR[VRB] is placed into halfword element i of VR[VRT]. This number ranges from 0 to 16, inclusive.

**Special Registers Altered:**  
None

#### Vector Count Leading Zeros Word VX-form

**Vclzw**  
VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1922</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```assembly
do i = 0 to 3
n ← 0
do while n < 32
  if VR[VRB].word[i].bit[n] = 0b1 then leave
  n ← n + 1
end
VSR[VRT].word[i] ← n
end
```

For each integer value i from 0 to 3, do the following.
A count of the number of consecutive zero bits starting at bit 0 of word element i of VR[VRB] is placed into word element i of VR[VRT]. This number ranges from 0 to 32, inclusive.

**Special Registers Altered:**  
None

#### Vector Count Leading Zeros Doubleword

**Vclzd**  
VRT,VRB

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
<th>1986</th>
</tr>
</thead>
</table>

if MSR.VEC=0 then Vector_Unavailable()

```assembly
do i = 0 to 1
n ← 0
do while (n<64) & (VR[VRB].dword[i].bit[n]=0b0)
  n ← n + 1
end
VSR[VRT].dword[i] ← n
end
```

For each integer value i from 0 to 1, do the following.
A count of the number of consecutive zero bits starting at bit 0 of doubleword element i of VR[VRB] is placed into doubleword element i of VR[VRT]. This number ranges from 0 to 64, inclusive.

**Special Registers Altered:**  
None
5.14 Vector Population Count Instructions

**Vector Population Count Byte**

\[ \text{vpopcntb } \text{VRT},\text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>1795</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do j = 0 to 7
      n ← n + VR[VRB].byte[i].bit[j]
   end
   VSR[VRT].byte[i] ← n
end

For each integer value \( i \) from 0 to 15, do the following. A count of the number of bits set to 1 in byte element \( i \) of \( VR[VRB] \) is placed into byte element \( i \) of \( VR[VRT] \). This number ranges from 0 to 8, inclusive.

**Special Registers Altered:** None

**Vector Population Count Doubleword**

\[ \text{vpopcntd } \text{VRT},\text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>1987</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do j = 0 to 63
      n ← n + VR[VRB].dword[i].bit[j]
   end
   VSR[VRT].dword[i] ← n
end

For each integer value \( i \) from 0 to 1, do the following. A count of the number of bits set to 1 in doubleword element \( i \) of \( VR[VRB] \) is placed into doubleword element \( i \) of \( VR[VRT] \). This number ranges from 0 to 64, inclusive.

**Special Registers Altered:** None

**Vector Population Count Halfword**

\[ \text{vpopcntb } \text{VRT},\text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>1859</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do j = 0 to 15
      n ← n + VR[VRB].hword[i].bit[j]
   end
   VSR[VRT].hword[i] ← n
end

For each integer value \( i \) from 0 to 7, do the following. A count of the number of bits set to 1 in halfword element \( i \) of \( VR[VRB] \) is placed into halfword element \( i \) of \( VR[VRT] \). This number ranges from 0 to 16, inclusive.

**Special Registers Altered:** None

**Vector Population Count Word**

\[ \text{vpopcntw } \text{VRT},\text{VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>///</th>
<th>VRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>1923</td>
<td></td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do j = 0 to 31
      n ← n + VR[VRB].word[i].bit[j]
   end
   VSR[VRT].word[i] ← n
end

For each integer value \( i \) from 0 to 3, do the following. A count of the number of bits set to 1 in word element \( i \) of \( VR[VRB] \) is placed into word element \( i \) of \( VR[VRT] \). This number ranges from 0 to 32, inclusive.

**Special Registers Altered:** None
5.15 Vector Bit Permute Instruction

Vector Bit Permute Quadword VX-form

\[ \text{vbpermq VRT,VRA,VRB} \]

<table>
<thead>
<tr>
<th></th>
<th>VRT</th>
<th>VRA</th>
<th>VRB</th>
<th>1356</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
    index ← VR[VRB].byte[i]
    if index < 128 then
        perm.bit[i] ← VR[VRA].bit[index]
    else
        perm.bit[i] ← 0
    end
VR[VRT].dword[0] ← Chop(EXTZ(perm),64)
VR[VRT].dword[1] ← 0x0000_0000_0000_0000

For each integer value \( i \) from 0 to 15, do the following.

Let \( \text{index} \) be the contents of byte element \( i \) of \( VR[VRB] \).

If \( \text{index} \) is less than 128, then the contents of bit \( i \) of \( VR[VRA] \) are placed into bit \( 48+i \) of double-word element \( i \) of \( VR[VRT] \). Otherwise, bit \( 48+i \) of doubleword element \( i \) of \( VR[VRT] \) is set to 0.

The contents of bits 0:47 of \( VR[VRT] \) are set to 0.
The contents of bits 63:127 of \( VR[VRT] \) are set to 0.

Special Registers Altered:
None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 127 permits the permuted bits to be selected from a 256-bit quantity, using a single index register. For example, assume that the 256-bit quantity \( Q \), from which the permuted bits are to be selected, is in registers \( v2 \) (high-order 128 bits of \( Q \)) and \( v3 \) (low-order 128 bits of \( Q \)), that the index values are in register \( v1 \), with each byte of \( v1 \) containing a value in the range 0:255, and that each byte of register \( v4 \) contains the value 128. The following code sequence selects eight permuted bits from \( Q \) and places them into the low-order byte of \( v6 \).

\[ \text{vbpermq v6,v1,v2} \quad \# \text{select from high-order half of } Q \]
\[ \text{xor v0,v1,v4} \quad \# \text{adjust index values} \]
\[ \text{vbpermq v5,v0,v3} \quad \# \text{select from low-order half of } Q \]
\[ \text{vor v6,v6,v5} \quad \# \text{merge the two selections} \]
5.16 Decimal Integer Arithmetic Instructions

The Decimal Integer Arithmetic instructions operate on decimal integer values only in signed packed decimal format. Signed packed decimal format consists of 31 4-bit base-10 digits of magnitude and a trailing 4-bit sign code. Operations are performed as sign-magnitude, and produce a decimal result placed in a Vector Register (i.e., \texttt{bcdadd}, \texttt{bcdsub}).

A valid encoding of a decimal integer value requires the following properties.
- Each of the 31 4-bit digits of the operand’s magnitude (bits 0:123) must be in the range 0-9.
- The sign code (bits 124:127) must be in the range 10-15.

Source operands with sign codes of \texttt{0b1010}, \texttt{0b1100}, \texttt{0b1110}, and \texttt{0b1111} are interpreted as positive values.

Source operands with sign codes of \texttt{0b1011} and \texttt{0b1101} are interpreted as negative values.

Positive and zero results are encoded with a either sign code of \texttt{0b1100} or \texttt{0b1111}, depending on the preferred sign (indicated as an immediate operand).

Negative results are encoded with a sign code of \texttt{0b1101}. 


Decimal Add Modulo VX-form

**bcadd.** VRT, VRA, VRB, PS

If MSR.VEC = 0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Add(VR[VRA], VR[VRB], PS)

**Table 5.1**

<table>
<thead>
<tr>
<th>4</th>
<th>VRT</th>
<th>6</th>
<th>VRA</th>
<th>16</th>
<th>VRB</th>
<th>22</th>
<th>23</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

src1 is added to src2.

If the unbounded result is equal to zero, do the following.
If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.
CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.
If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.
If the operation overflows, CR field 6 is set to 0b1010. Otherwise, CR field 6 is set to 0b1000.

If the unbounded result is less than zero, do the following.
The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 0b1001. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VR[VRT].
The sign code is placed in bits 124:127 of VR[VRT].

If either src1 or src2 is an invalid encoding of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:
CR field 6

---

Decimal Subtract Modulo VX-form

**bcdsub.** VRT, VRA, VRB, PS

If MSR.VEC = 0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Subtract(VR[VRA], VR[VRB], PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

src1 is subtracted by src2.

If the unbounded result is equal to zero, do the following.
If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.
CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.
If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.
If the operation overflows, CR field 6 is set to 0b1010. Otherwise, CR field 6 is set to 0b1000.

If the unbounded result is less than zero, do the following.
The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 0b1001. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VR[VRT].
The sign code is placed in bits 124:127 of VR[VRT].

If either src1 or src2 is an invalid encoding of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:
CR field 6
5.17 Vector Status and Control Register Instructions

*Move To Vector Status and Control Register VX-form*

```markdown
mtvscr VRB
```

The contents of word element 3 of VRB are placed into the VSCR.

**Special Registers Altered:**
None

*Move From Vector Status and Control Register VX-form*

```markdown
mfvscr VRT
```

The contents of the VSCR are placed into word element 3 of VRT.

The remaining word elements in VRT are set to 0.

**Special Registers Altered:**
None
Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point]

6.1 Decimal Floating-Point (DFP) Facility Overview

This chapter describes the behavior of the decimal floating-point facility, the supported data types, formats, and classes, and the usage of registers. Also included are the execution model, exceptions, and instructions supported by the decimal floating-point facility.

The decimal floating-point (DFP) facility shares the 32 floating-point registers (FPRs) and the Floating-Point Status and Control Register (FPSCR) with the floating-point (BFP) facility. However, the interpretation of data formats in the FPRs, and the meaning of some control and status bits in the FPSCR are different between the BFP and DFP facilities.

The DFP facility also shares the Condition Register (CR) with the fixed-Point facility, the BFP facility, and the vector facility.

The DFP facility supports three DFP data formats: DFP Short (single precision), DFP Long (double precision), and DFP Extended (quad precision). Most operations are performed on DFP Long or DFP Extended format directly. Support for DFP Short is limited to conversion to and from DFP Long. Some DFP instructions operate on other data types, including signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

DFP instructions are provided to perform arithmetic, compare, test, quantum-adjustment, conversion, and format operations on operands held in FPRs or FPR pairs.

- **Arithmetic instructions**
  These instructions perform addition, subtraction, multiplication, and division operations.

- **Compare instructions**
  These instructions perform a comparison operation on the numerical value of two DFP operands.

- **Test instructions**
  These instructions test the data class, the data group, the exponent, or the number of significant digits of a DFP operand.

- **Quantum-adjustment instructions**
  These instructions convert a DFP number to a result in the form that has the designated exponent, which may be explicitly or implicitly specified.

- **Conversion instructions**
  These instructions perform conversion between different data formats or data types.

- **Format instructions**
  These instructions facilitate composing or decomposing a DFP operand.

These instructions are described in Section 6.6 “DFP Instruction Descriptions” on page 288.

The three DFP data formats allow finite numbers to be represented with different precision and ranges. Special codes are also provided to represent +Infinity, -Infinite, Quiet NaN (Not-a-Number), and Signaling NaN. Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. The encoding of NaNs provides a diagnostic information field. This diagnostic field may be used to indicate such things as the source of an uninitialized variable or the reason an invalid result was produced.

The DFP processor recognizes a set of DFP exceptions which are indicated via bits set in the FPSCR. Additionally, the DFP exception actions depend on the setting of the various exception enable bits in the FPSCR.

The following DFP exceptions are detected by the DFP processor. The exception status bits in the FPSCR are indicated in parentheses.

- **Invalid Operation Exception** (VX)
  SNaN (VXSNAN)
  $\infty \times \infty$ (VXISI)
  $\infty \div \infty$ (VXIDI)
  $0 \div 0$ (VXZDZ)
Each DFP exception and each category of Invalid Operation Exception has an exception status bit in the FPSCR. In addition, each of the five DFP exceptions has a corresponding enable bit in the FPSCR. These enable bits enable or disable the invocation of the system floating-point enabled exception error handler, and may affect the setting of some exception status bits in the FPSCR.

The usage of these bits by the DFP facility differs from the usage by the BFP facility. Section 6.5.10 “DFP Exceptions” on page 280 provides a detailed discussion of DFP exceptions, including the effects of the enable bits.

6.2 DFP Register Handling

The following sections describe first how the floating-point registers are utilized by the DFP facility. The subsequent section covers the DFP usage of CR and FPSCR.

6.2.1 DFP Usage of Floating-Point Registers

The DFP facility shares the same 32 64-bit FPRs with the BFP facility. Like the FP instructions, DFP instructions also use 5-bit fields for designating the FPRs to hold the source or target operands.

When data in DFP Short format is held in a FPR, it occupies the rightmost 32 bits of the FPR. The Load Floating-Point as Integer Word Algebraic instruction is provided to load the rightmost 32 bits of a FPR with a single-word data from storage. The Store Floating-Point as Integer Word instruction is available to store the rightmost 32 bits of a FPR to a storage location.

Data in DFP Long format, 64-bit binary fixed-point values, or 64-bit BCD values is held in a FPR using all 64 bits. Data of 64 bits may be loaded from storage via any of the Load Floating-Point Double instructions and stored via any of the Store Floating-Point Double instructions.

Data in DFP Extended format or 128-bit BCD values is held in an even-odd FPR pair using all 128 bits. Data of 128 bits must be loaded into the desired even-odd pair of floating-point registers using an appropriate sequence of the Load Floating-Point Double instructions and stored using an appropriate sequence of the Store Floating-Point Double instructions.

Data used as a source operand by any Decimal Floating-Point instruction that was produced, either directly or indirectly, by a Load Floating-Point Single instruction, a Floating Round to Single-Precision instruction, or a binary floating-point single-precision arithmetic instruction is boundedly undefined.

When an even-odd FPR pair is used to hold a 128-bit operand, the even-numbered FPR is used to hold the leftmost doubleword of the operand and the next higher-numbered FPR is used to hold the rightmost doubleword. A DFP instruction designating an odd-numbered FPR for a 128-bit operand is an invalid instruction form.

Programming Note

The Floating-Point Move instructions can be used to move operands between FPRs.

The bit definitions for the FPSCR are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:28</td>
<td>Reserved</td>
</tr>
<tr>
<td>29:31</td>
<td>DFP Rounding Control (DRN)</td>
</tr>
<tr>
<td></td>
<td>See Section 6.5.2, “Rounding Mode Specification” on page 277.</td>
</tr>
<tr>
<td></td>
<td>000 Round to Nearest, Ties to Even</td>
</tr>
<tr>
<td></td>
<td>001 Round toward Zero</td>
</tr>
<tr>
<td></td>
<td>010 Round toward +Infinity</td>
</tr>
<tr>
<td></td>
<td>011 Round toward -Infinity</td>
</tr>
<tr>
<td></td>
<td>100 Round to Nearest, Ties away from 0</td>
</tr>
<tr>
<td></td>
<td>101 Round to Nearest, Ties toward 0</td>
</tr>
<tr>
<td></td>
<td>110 Round to away from Zero</td>
</tr>
<tr>
<td></td>
<td>111 Round to Prepare for Shorter Precision</td>
</tr>
</tbody>
</table>

Programming Note

FPSCR28 is reserved for extension of the DRN field, therefore DRN may be set using the mtfsf1 instruction to set the rounding mode.

Floating-Point Exception Summary (FX)
Every floating-point instruction, except mtfsfi and mtfsf, implicitly sets FPSCRFX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. mcrfs, mtfsfi, mtfsf, mtfsf0, and mtfsf1 can alter FPSCRFX explicitly.

Floating-Point Enabled Exception Summary (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. mcrfs, mtfsfi, mtfsf, mtfsf0, and mtfsf1 cannot alter FPSCRFX explicitly.

Floating-Point Invalid Operation Exception Summary (VX)
This bit is the OR of all the Invalid Operation
exception bits. \textit{mcrfs}, \textit{mtfsfi}, \textit{mtfsf}, \textit{mtfsb0}, and \textit{mtfsb1} cannot alter \textit{FPSCRXX} explicitly.

35 \textbf{Floating-Point Overflow Exception} (OX) See Section 6.5.10.3, “Overflow Exception” on page 283.

36 \textbf{Floating-Point Underflow Exception} (UX) See Section 6.5.10.4, “Underflow Exception” on page 284.

37 \textbf{Floating-Point Zero Divide Exception} (ZX) See Section 6.5.10.2, “Zero Divide Exception” on page 283.

38 \textbf{Floating-Point Inexact Exception} (XX) See Section 6.5.10.5, “Inexact Exception” on page 285.

FPSCRXX is a sticky version of FPSCRFI (see below). Thus the following rules completely describe how FPSCRXX is set by a given instruction.

- If the instruction affects FPSCRFI, the new value of FPSCRXX is obtained by ORing the old value of FPSCRXX with the new value of FPSCRFI.
- If the instruction does not affect FPSCRFI, the value of FPSCRXX is unchanged.

39 \textbf{Floating-Point Invalid Operation Exception (SNaN)} (VXSNAN) See Section 6.5.10.1, “Invalid Operation Exception” on page 282.

40 \textbf{Floating-Point Invalid Operation Exception (∞ - ∞)} (VXISI) See Section 6.5.10.1.

41 \textbf{Floating-Point Invalid Operation Exception (∞ + ∞)} (VXIDI) See Section 6.5.10.1.

42 \textbf{Floating-Point Invalid Operation Exception (0/0)} (VXZDZ) See Section 6.5.10.1.

43 \textbf{Floating-Point Invalid Operation Exception (∞ × 0)} (VXMIZ) See Section 6.5.10.1.

44 \textbf{Floating-Point Invalid Operation Exception (Invalid Compare)} (VXVC) See Section 6.5.10.1.

45 \textbf{Floating-Point Fraction Rounded} (FR) The last Arithmetic or Rounding and Conversion instruction incremented the fraction during rounding. See Section 6.5.1, “Rounding” on page 276. This bit is not sticky.

46 \textbf{Floating-Point Fraction Inexact} (FI) The last Arithmetic or Rounding and Conversion instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 6.5.1. This bit is not sticky.

47:51 \textbf{Floating-Point Result Flags} (FPRF)

- \textbf{Floating-Point Result Class Descriptor} (C)
  Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 71 on page 272.

48:51 \textbf{Floating-Point Condition Code} (FPCC)

- \textbf{Floating-Point Less Than or Negative} (FL or <)
- \textbf{Floating-Point Greater Than or Positive} (FG or >)
- \textbf{Floating-Point Equal or Zero} (FE or =)
- \textbf{Floating-Point Unordered or NaN} (FU or ?)

52 Reserved

53 \textbf{Floating-Point Invalid Operation Exception (Software Request)} (VXSOFT)
This bit can be altered only by \textit{mcrfs}, \textit{mtfsfi}, \textit{mtfsf}, \textit{mtfsb0}, or \textit{mtfsb1}. See Section 6.5.10.1, “Invalid Operation Exception” on page 282.

Neither used nor changed by DFP.

\textbf{Programming Note}

Although the architecture does not provide a DFP square root instruction, if software simulates such an instruction, it should set bit 54 whenever the source operand of the square root function is invalid.
Floating-Point Invalid Operation Exception Enable (VE)  
See Section 6.5.10.1.

Floating-Point Overflow Exception Enable (OE)  
See Section 6.5.10.3, “Overflow Exception” on page 283.

Floating-Point Underflow Exception Enable (UE)  
See Section 6.5.10.4, “Underflow Exception” on page 284.

Floating-Point Zero Divide Exception Enable (ZE)  
See Section 6.5.10.2, “Zero Divide Exception” on page 283.

Floating-Point Inexact Exception Enable (XE)  
See Section 6.5.10.5, “Inexact Exception” on page 285.

Reserved (not used by DFP)

Binary Floating-Point Rounding Control (RN)  
See Section 6.5.1, “Rounding” on page 276.

<table>
<thead>
<tr>
<th>Result Flags</th>
<th>Result Value Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = ?</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>Signaling NaN (DFP only)</td>
</tr>
<tr>
<td>1 0 0 0 1</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>- Infinity</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>- Normal Number</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>- Subnormal Number</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>- Zero</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>+ Zero</td>
</tr>
<tr>
<td>1 0 1 0 0</td>
<td>+ Subnormal Number</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>+ Normal Number</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>+ Infinity</td>
</tr>
</tbody>
</table>

Figure 71. Floating-Point Result Flags

6.3 DFP Support for Non-DFP Data Types

In addition to the DFP data types, the DFP processor provides limited support for the following non-DFP data types: signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

In unsigned binary fixed-point data, all bits are used to express the absolute value of the number. For signed binary fixed-point data, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two's complement binary notation with a one in the sign-bit position.

For decimal data, each byte contains a pair of four-bit nibbles; each four-bit nibble contains a binary-coded-decimal (BCD) code. There are two kinds of BCD codes: digit code and sign code. For unsigned decimal data, all nibbles contain a digit code (D) as shown in Figure 72.

```
D D D D ... D D D D
```

Figure 72. Format for Unsigned Decimal Data

For signed decimal data, the rightmost nibble contains a sign code (S) and all other nibbles contain a digit code as shown in Figure 73.

```
D D D D ... D D D D S
```

Figure 73. Format for Signed Decimal Data

The decimal digits 0-9 have the binary encoding 0000-1001. The preferred plus-sign codes are 1100 and 1111. The preferred minus sign code is 1101. These are the sign codes generated for the results of the `Decode DPD To BCD` instruction. A selection is provided by this instruction to specify which of the two preferred plus sign codes is to be generated. Alternate sign codes are also recognized as valid in the sign position: 1010 and 1110 are alternate sign codes for plus, and 1011 is an alternate sign code for minus. Alternate sign codes are accepted for any source operand, but are not generated as a result by the instruction. When an invalid digit or sign code is detected by the `Encode BCD To DPD` instruction, an invalid-opera-
tion exception occurs. A summary of digit and sign codes are provided in Figure 74.

<table>
<thead>
<tr>
<th>Binary Code</th>
<th>Recognized As</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0  Invalid</td>
</tr>
<tr>
<td>0001</td>
<td>1  Invalid</td>
</tr>
<tr>
<td>0010</td>
<td>2  Invalid</td>
</tr>
<tr>
<td>0011</td>
<td>3  Invalid</td>
</tr>
<tr>
<td>0100</td>
<td>4  Invalid</td>
</tr>
<tr>
<td>0101</td>
<td>5  Invalid</td>
</tr>
<tr>
<td>0110</td>
<td>6  Invalid</td>
</tr>
<tr>
<td>0111</td>
<td>7  Invalid</td>
</tr>
<tr>
<td>1000</td>
<td>8  Invalid</td>
</tr>
<tr>
<td>1001</td>
<td>9  Invalid</td>
</tr>
<tr>
<td>1010</td>
<td>Invalid</td>
</tr>
<tr>
<td>1011</td>
<td>Plus</td>
</tr>
<tr>
<td>1100</td>
<td>Plus (preferred; option 1)</td>
</tr>
<tr>
<td>1101</td>
<td>Minus (preferred)</td>
</tr>
<tr>
<td>1110</td>
<td>Plus</td>
</tr>
<tr>
<td>1111</td>
<td>Plus (preferred; option 2)</td>
</tr>
</tbody>
</table>

Figure 74. Summary of BCD Digit and Sign Codes

### 6.4 DFP Number Representation

A DFP finite number consists of three components: a sign bit, a signed exponent, and a significand. The signed exponent is a signed binary integer. The significand consists of a number of decimal digits, which are to the left of the implied decimal point. The rightmost digit of the significand is called the units digit. The numerical value of a DFP finite number is represented as $(-1)^{sign} \times significand \times 10^{exponent}$, and the unit value of this number is $(1 \times 10^{exponent})$, which is called the quantum.

DFP finite numbers are not normalized. This allows leading zeros and trailing zeros to exist in the significand. This unnormalized DFP number representation allows some values to have redundant forms; each form represents the DFP number with a different combination of the significand value and the exponent value. For example, $1000000 \times 10^5$ and $10 \times 10^{10}$ are two different forms of the same numerical value. A form of this number representation carries information about both the numerical value and the quantum of a DFP finite number.

The significant digits of a DFP finite number are the digits in the significand beginning with the leftmost non-zero digit and ending with the units digit.

### 6.4.1 DFP Data Format

DFP numbers and NaNs may be represented in FPRs in any of the three data formats: DFP Short, DFP Long, or DFP Extended. The contents of each data format represent encoded information. Special codes are assigned to NaNs and infinities. Different formats support different sizes in both significand and exponent. Arithmetic, compare, test, quantum-adjustment, and format instructions are provided for DFP Long and DFP Extended formats only.

The sign is encoded as a one bit binary value. Significand is encoded as an unsigned decimal integer in two distinct parts. The leftmost digit (LMD) of the significand is encoded as part of the combination field; the remaining digits of the significand are encoded in the trailing significand field. The exponent is contained in the combination field in two parts. However, prior to encoding, the exponent is converted to an unsigned binary value called the biased exponent by adding a bias value which is a constant for each format. The two leftmost bits of the biased exponent are encoded with the leftmost digit of the significand in the leftmost bits of the combination field. The rest of the biased exponent occupies the remaining portion of the combination field.

### 6.4.1.1 Fields Within the Data Format

The DFP data representation comprises three fields, as diagrammed below for each of the three formats:

**DFP Short format**

```
  S  G  T
0  1  12 31
```

Figure 75. DFP Short format

**DFP Long format**

```
  S  G  T
0  1  14 63
```

Figure 76. DFP Long format

**DFP Extended format**

```
  S  G  T (continued)
0  1  18 63
```

Figure 77. DFP Extended format

The fields are defined as follows:

**Sign bit (S)**
The sign bit is in bit 0 of each format, and is zero for plus and one for minus.

**Combination field (G)**
As the name implies, this field provides a combination of the exponent and the left-most digit (LMD) of the significand, for finite numbers, or provides a special code
for denoting the value as either a Not-a-Number or an Infinity.

The first 5 bits of the combination field contain the encoding of NaN or infinity, or the two leftmost bits of the biased exponent and the leftmost digit (LMD) of the significand. The following tables show the encoding:

<table>
<thead>
<tr>
<th>G0:4</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111</td>
<td>NaN</td>
</tr>
<tr>
<td>11110</td>
<td>Infinity</td>
</tr>
<tr>
<td>All others</td>
<td>Finite Number (see Figure 79)</td>
</tr>
</tbody>
</table>

**Figure 78. Encoding of the G field for Special Symbols**

<table>
<thead>
<tr>
<th>LMD</th>
<th>Leftmost 2-bits of biased exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000   01000   10000</td>
</tr>
<tr>
<td>1</td>
<td>00001   01001   10001</td>
</tr>
<tr>
<td>2</td>
<td>00010   01010   10010</td>
</tr>
<tr>
<td>3</td>
<td>00011   01011   10011</td>
</tr>
<tr>
<td>4</td>
<td>00100   01100   10100</td>
</tr>
<tr>
<td>5</td>
<td>00101   01101   10101</td>
</tr>
<tr>
<td>6</td>
<td>00110   01110   10110</td>
</tr>
<tr>
<td>7</td>
<td>00111   01111   10111</td>
</tr>
<tr>
<td>8</td>
<td>11000   11010   11100</td>
</tr>
<tr>
<td>9</td>
<td>11001   11011   11101</td>
</tr>
</tbody>
</table>

**Figure 79. Encoding of bits 0:4 of the G field for Finite Numbers**

For DFP finite numbers, the rightmost N-5 bits of the N-bit combination field contain the remaining bits of the *biased exponent*. For NaNs, bit 5 of the combination field is used to distinguish a Quiet NaN from a Signaling NaN; the remaining bits in a source operand are ignored and they are set to zeros in a target operand by most operations. For infinities, the rightmost N-5 bits of the N-bit combination field of a source operand are ignored and they are set to zeros in a target operand by most operations.

**Trailing Significand field** (T)

For DFP finite numbers, this field contains the remaining *significand* digits. For NaNs, this field may be used to contain diagnostic information. For infinities, contents in this field of a source operand are ignored and they are set to zeros in a target operand by most operations. The trailing significand field is a multiple of 10-bit blocks. The multiple depends on the format. Each 10-bit block is called a declet and represents three decimal digits, using the Densely Packed Decimal (DPD) encoding defined in Appendix B.

### 6.4.1.2 Summary of DFP Data Formats

The properties of the three DFP formats are summarized in the following table:

<table>
<thead>
<tr>
<th>Format</th>
<th></th>
<th>DFP Short</th>
<th>DFP Long</th>
<th>DFP Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Widths (bits):</td>
<td></td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Format</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sign (S)</td>
<td></td>
<td>11</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>Combination (G)</td>
<td></td>
<td>20</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>Trailing Significand (T)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent:</td>
<td></td>
<td>191</td>
<td>767</td>
<td>12,287</td>
</tr>
<tr>
<td>Maximum biased</td>
<td></td>
<td>90</td>
<td>369</td>
<td>6111</td>
</tr>
<tr>
<td>Maximum (Xmax)</td>
<td></td>
<td>-101</td>
<td>-398</td>
<td>-6176</td>
</tr>
<tr>
<td>Minimum (Xmin)</td>
<td></td>
<td>101</td>
<td>398</td>
<td>6176</td>
</tr>
<tr>
<td>Bias</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision (p) (digits)</td>
<td></td>
<td>7</td>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>Magnitude:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum normal number (Nmax)</td>
<td></td>
<td>(10^7 - 1) x 10^90</td>
<td>(10^16 - 1) x 10^369</td>
<td>(10^34 - 1) x 10^6111</td>
</tr>
<tr>
<td>Minimum normal number (Nmin)</td>
<td></td>
<td>1 x 10^-95</td>
<td>1 x 10^-383</td>
<td>1 x 10^-6143</td>
</tr>
</tbody>
</table>
6.4.1.3 Preferred DPD Encoding

Execution of DFP instructions decodes source operands from DFP data formats to an internal format for processing, and encodes the operation result before the final result is returned as the target operand.

As part of the decoding process, declets in the trailing significand field of source operands are decoded to their corresponding BCD digit codes using the DPD-to-BCD decoding algorithm. As part of the encoding process, BCD digit codes to be stored into the trailing significand field of the target operand are encoded into declets using the BCD-to-DPD encoding algorithm. Both the decoding and encoding algorithms are defined in Appendix B.

As explained in Appendix B, there are eight 3-digit decimal values that have redundant DPD codes and one preferred DPD code. All redundant DPD codes are recognized in source operands for the associated 3-digit decimal number. DFP operations will always generate the preferred DPD codes for the trailing significand field of the target operand.

6.4.2 Classes of DFP Data

There are six classes of DFP data, which include numerical and nonnumeric entities. The numerical entities include zero, subnormal number, normal number, and infinity data classes. The nonnumeric entities include quiet and signaling NaNs data classes. The value of a DFP finite number, including zero, subnormal number, and normal number, is a quantization of the real number based on the data format. The Test Data Class instruction may be used to determine the class of a DFP operand. In general, an operation that returns a DFP result sets the FPCR.FPRF field to indicate the data class of the result.

The following tables show the value ranges for finite-number data classes, and the codes for NaNs and infinities.

<table>
<thead>
<tr>
<th>Format</th>
<th>DFP Short</th>
<th>DFP Long</th>
<th>DFP Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum subnormal number ($D_{\text{min}}$)</td>
<td>$1 \times 10^{-101}$</td>
<td>$1 \times 10^{-398}$</td>
<td>$1 \times 10^{-6176}$</td>
</tr>
</tbody>
</table>

Figure 80. Summary of DFP Formats
Zeros have a zero significand and any representable value in the exponent. A +0 is distinct from -0, and zeros with different exponents are distinct, except that comparison treats them as equal.

Subnormal Numbers
Subnormal numbers have values that are smaller than $N_{\text{min}}$ and greater than zero in magnitude.

Normal Numbers
Normal numbers are nonzero finite numbers whose magnitude is between $N_{\text{min}}$ and $N_{\text{max}}$ inclusively.

Infinities
Infinities are represented by 0b11110 in the leftmost 5 bits of the combination field. When an operation is defined to generate an infinity as the result, a default infinity is sometimes supplied. A default infinity has all remaining bits in the combination field and trailing significand field set to zeros.

When infinities are used as source operands, only the leftmost 5 bits of the combination field are interpreted (i.e., 0b11110 indicates the value is an infinity). The trailing significand field of infinities is usually ignored. For generated infinities, the leftmost 5 bits of the combination field are set to 0b11110 and all remaining combination bits are set to zero.

Infinities can participate in most arithmetic operations and give a consistent result. In comparisons, any +Infinity compares greater than any finite number, and any -Infinity compares less than any finite number. All +Infinity are compared equal and all -Infinity are compared equal.

Signaling and Quiet NaNs
There are two types of Not-a-Numbers (NaNs), Signaling (SNaN) and Quiet (QNaN).

A special QNaN is sometimes supplied as the default QNaN for a disabled invalid-operation exception; it has a plus sign, the leftmost 6 bits of the combination field set to 0b111110 and remaining bits in the combination field and the trailing significand field set to zero.

Normally, source QNaNs are propagated during operations so that they will remain visible at the end. When a QNaN is propagated, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost N-6 bits of the combination field set to zero, where N is the width of the combination field for the format.

A source SNaN generally causes an invalid-operation exception. If the exception is disabled, the SNaN is converted to the corresponding QNaN and propagated. The primary encoding difference between an SNaN and a QNaN is that bit 5 of an SNaN is 1 and bit 5 of a QNaN is 0. When an SNaN is propagated as a QNaN, bit 5 is set to 0, and, just as with QNaN propagation, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost N-6 bits of the combination field set to zero, where N is the width of the combination field for the format. For some format-conversion instructions, a source SNaN does not cause an invalid-operation exception, and an SNaN is returned as the target operand.

For instructions with two source NaNs and a NaN is to be propagated as the result, do the following:
- If there is a QNaN in FRA and an SNaN in FRB, the SNaN in FRB is propagated.
- Otherwise, propagate the NaN in FRA.

6.5 DFP Execution Model

DFP operations are performed as if they first produce an intermediate result correct to infinite precision and with unbounded range. The intermediate result is then rounded to the destination's precision according to one of the eight DFP rounding modes. If the rounded result has only one form, it is delivered as the final result; if the rounded result has redundant forms, then an ideal exponent is used to select the form of the final result. The ideal exponent determines the form, not the value, of the final result. (See Section 6.5.3 "Formation of Final Result" on page 278.)

6.5.1 Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's precision. The destination's precision of an operation defines the set of permissible resultant values. For
most operations, the destination’s precision is the target-format precision and the permissible resultant values are those values representable in the target format. For some special operations, the destination precision is constrained by both the target format and some additional restrictions, and the permissible resultant values are a subset of the values representable in the target format.

Rounding sets FPSCR bits FR and FI. When an inexact exception occurs, FI is set to one; otherwise, FI is set to zero. When an inexact exception occurs and if the rounded result is greater in magnitude than the intermediate result, then FR is set to one; otherwise, FR is set to zero. The exception is the Round to FP Integer Without Inexact instruction, which always sets FR and FI to zero. Rounding may cause an overflow exception or underflow exception; it may also cause an inexact exception.

Refer to Figure 83 below for rounding. Let Z be the intermediate result of a DFP operation. Z may or may not fit in the destination’s precision. If Z is exactly one of the permissible representable resultant values, then the final result in all rounding modes is Z. Otherwise, either Z1 or Z2 is chosen to approximate the result, where Z1 and Z2 are the next larger and smaller permissible resultant values, respectively.

![Figure 83. Rounding](image)

**Round to Nearest, Ties to Even**
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the larger in magnitude (Z1 or Z2).

**Round to Nearest, Ties toward 0**
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the smaller in magnitude (Z1 or Z2).

**Round toward +∞**
Choose Z1.

**Round toward -∞**
Choose Z2.

**Round to Nearest, Ties away from 0**
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the larger in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude at least \((N_{\text{max}} + 0.5Q(N_{\text{max}}))\) is rounded to infinity with no change in sign; where \(Q(N_{\text{max}})\) is the quantum of \(N_{\text{max}}\).

**Round to FP Integer**
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the smaller in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude greater than \((N_{\text{max}} + 0.5Q(N_{\text{max}}))\) is rounded to infinity with no change in sign; where \(Q(N_{\text{max}})\) is the quantum of \(N_{\text{max}}\).

**Round away from 0**
Choose the larger in magnitude (Z1 or Z2).

**Round to prepare for shorter precision**
Choose the smaller in magnitude (Z1 or Z2). If the selected value is inexact and the units digit of the selected value is either 0 or 5, then the digit is incremented by one and the incremented result is delivered. In all other cases, the selected value is delivered. When a value has redundant forms, the units digit is determined by using the form that has the smallest exponent.

### 6.5.2 Rounding Mode Specification

Unless otherwise specified in the instruction definition, the rounding mode used by an operation is specified in the DFP rounding control (DRN) field of the FPSCR. The eight DFP rounding modes are encoded in the DRN field as specified in the table below.

<table>
<thead>
<tr>
<th>DRN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Round to Nearest, Ties to Even</td>
</tr>
<tr>
<td>001</td>
<td>Round toward 0</td>
</tr>
<tr>
<td>010</td>
<td>Round toward +Infinity</td>
</tr>
<tr>
<td>011</td>
<td>Round toward -Infinity</td>
</tr>
<tr>
<td>100</td>
<td>Round to Nearest, Ties away from 0</td>
</tr>
<tr>
<td>101</td>
<td>Round to Nearest, Ties toward 0</td>
</tr>
<tr>
<td>110</td>
<td>Round away from 0</td>
</tr>
<tr>
<td>111</td>
<td>Round to Prepare for Shorter Precision</td>
</tr>
</tbody>
</table>

**Figure 84. Encoding of DFP Rounding-Mode Control (DRN)**

For the quantum-adjustment, a 2-bit immediate field, called RMC (Rounding Mode Control), in the instruction specifies the rounding mode used. The RMC field may contain a primary encoding or a secondary encoding. For Quantize, Quantize Immediate, and Reround, the RMC field contains the primary encoding. For Round to FP Integer the field contains either encoding, depending on the setting of a RMC-encoding-selection.
The following tables define the primary encoding and the secondary encoding.

**Figure 85. Primary Encoding of Rounding-Mode Control**

<table>
<thead>
<tr>
<th>Primary Rounding Mode</th>
<th>RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to nearest, ties to even</td>
</tr>
<tr>
<td>01</td>
<td>Round toward 0</td>
</tr>
<tr>
<td>10</td>
<td>Round to nearest, ties away from 0</td>
</tr>
<tr>
<td>11</td>
<td>Round according to FPSCRDRN</td>
</tr>
</tbody>
</table>

**Figure 86. Secondary Encoding of Rounding-Mode Control**

<table>
<thead>
<tr>
<th>Secondary Rounding Mode</th>
<th>RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to $+\infty$</td>
</tr>
<tr>
<td>01</td>
<td>Round to $-\infty$</td>
</tr>
<tr>
<td>10</td>
<td>Round away from 0</td>
</tr>
<tr>
<td>11</td>
<td>Round to nearest, ties toward 0</td>
</tr>
</tbody>
</table>

**6.5.3 Formation of Final Result**

An ideal exponent is defined for each DFP instruction that returns a DFP data operand.

**6.5.3.1 Use of Ideal Exponent**

For all DFP operations,

- if the rounded intermediate result has only one form, then that form is delivered as the final result.
- if the rounded intermediate result has redundant forms and is exact, then the form with the exponent closest to the ideal exponent is delivered.
- if the rounded intermediate result has redundant forms and is inexact, then the form with the smallest exponent is delivered.

The following table specifies the ideal exponent for each instruction.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Ideal Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>min(E(FRA), E(FRB))</td>
</tr>
<tr>
<td>Subtract</td>
<td>min(E(FRA), E(FRB))</td>
</tr>
<tr>
<td>Multiply</td>
<td>E(FRA) + E(FRB)</td>
</tr>
<tr>
<td>Divide</td>
<td>E(FRA) - E(FRB)</td>
</tr>
<tr>
<td>Quantize-Immediate</td>
<td>See Instruction Description</td>
</tr>
<tr>
<td>Quantize</td>
<td>E(FRA)</td>
</tr>
<tr>
<td>Reround</td>
<td>See Instruction Description</td>
</tr>
<tr>
<td>Round to FP Integer</td>
<td>max(0, E(FRA))</td>
</tr>
<tr>
<td>Convert to DFP Long</td>
<td>E(FRA)</td>
</tr>
<tr>
<td>Convert to DFP Extended</td>
<td>E(FRA)</td>
</tr>
<tr>
<td>Round to DFP Short</td>
<td>E(FRA)</td>
</tr>
<tr>
<td>Round to DFP Long</td>
<td>E(FRA)</td>
</tr>
<tr>
<td>Convert from Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Encode BCD to DPD</td>
<td>0</td>
</tr>
<tr>
<td>Insert Biased Exponent</td>
<td>E(FRA)</td>
</tr>
</tbody>
</table>

Notes:

- E(x) - exponent of the DFP operand in register x.

**Figure 87. Summary of Ideal Exponents**

**6.5.4 Arithmetic Operations**

Four arithmetic operations are provided: Add, Subtract, Multiply, and Divide.

**6.5.4.1 Sign of Arithmetic Result**

The following rules govern the sign of an arithmetic operation when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the source operand having the larger absolute value. If both source operands have the same sign, the sign of the result of an add operation is the same as the sign of the source operands. When the sum of two operands with opposite signs is exactly zero, the sign of the result is positive in all rounding modes except Round toward $-\infty$, in which case the sign is negative.

- The sign of the result of the subtract operation $x - y$ is the same as the sign of the result of the add operation $x + (-y)$.

- The sign of the result of a multiply or divide operation is the exclusive-OR of the signs of the source operands.
6.5.5 Compare Operations

Two sets of instructions are provided for comparing numerical values: Compare Ordered and Compare Unordered. In the absence of NaNs, these instructions work the same. These instructions work differently when either of the followings is true:

1. At least one source operand of the instruction is an SNaN and the invalid-operation exception is disabled.
2. When there is no SNaN in any source operand, at least one source operand of the instruction is a QNaN.

In case 1, Compare Unordered recognizes an invalid-operation exception and sets the FPSCRVXSNAN flag, but Compare Ordered recognizes the exception and sets both the FPSCRVXSNAN and FPSCRVXVC flags. In case 2, Compare Unordered does not recognize an exception, but Compare Ordered recognizes an invalid-operation exception and sets the FPSCRVXVC flag.

For finite numbers, comparisons are performed on values, that is, all redundant forms of a DFP number are treated equal.

Comparisons are always exact and cannot cause an inexact exception.

Comparison ignores the sign of zero, that is, +0 equals -0.

Infinities with like sign compare equal, that is, +∞ equals +∞, and -∞ equals -∞.

A NaN compares as unordered with any other operand, whether a finite number, an infinity, or another NaN, including itself.

Execution of a compare instruction always completes, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

6.5.6 Test Operations

Four kinds of test operations are provided: Test Data Class, Test Data Group, Test Exponent, and Test Significance.

The Test Data Class instruction examines the contents of a source operand and determines if the operand is one of the specified data classes. The test result and the sign of the source operand are indicated in the FPSCRFPCC field and CR field BF.

The Test Data Group instruction examines the contents of a source operand and determines if the operand is one of the specified data groups. The test result and the sign of the source operand are indicated in the FPSCRFPCC field and CR field BF.

The Test Exponent instruction compares the exponent of the two source operands. The test operation ignores the sign and significand of operands. Infinities compare equal, and NaNs compare equal. The test result is indicated in the FPSCRFPCC field and CR field BF.

The Test Significance instruction compares the number of significant digits of one source operand with the referenced number of significant digits in another source operand. The test result is indicated in the FPSCRFPCC field and CR field BF.

Execution of a test instruction does not cause any DFP exception.

6.5.7 Quantum Adjustment Operations

Four kinds of quantum-adjustment operations are provided: Quantize, Quantize Immediate, Reround, and Round To FP Integer. Each of them has an immediate field which specifies whether the rounding mode in FPSCR or a different one is to be used.

The Quantize instruction is used to adjust a DFP number to the form that has the specified target exponent. The Quantize Immediate instruction is similar to the Quantize instruction, except that the target exponent is specified in a 5-bit immediate field as a signed binary integer and has a limited range.

The Reround instruction is used to simulate a DFP operation of a precision other than that of DFP Long or DFP Extended. For the Reround instruction to produce a result which accurately reflects that which would have resulted from a DFP operation of the desired precision d in the range {1: 33} inclusively, the following conditions must be met:

- The precision of the preceding DFP operation must be at least one digit larger than d.
- The rounding mode used by the preceding DFP operation must be round-to-prepare-for-shorter-precision.

The Round To FP Integer instruction is used to round a DFP number to an integer value of the same format. The target exponent is implicitly specified, and is greater than or equal to zero.

6.5.8 Conversion Operations

There are two kinds of conversion operations: data-format conversion and data-type conversion.

6.5.8.1 Data-Format Conversion

The instructions Convert To DFP Long and Convert To DFP Extended convert DFP operands to wider formats; the instructions Round To DFP Short and Round To DFP Long convert DFP operands to narrower formats.

When converting a finite number to a wider format, the result is exact. When converting a finite number to a
narrower format, the source operand is rounded to the
target-format precision, which is specified by the
instruction, not by the target register size.

When converting a finite number, the ideal exponent of
the result is the source exponent.

Conversion of an infinity or NaN to a different format
does not preserve the source combination field. Let N
be the width of the target format's combination field.

- When the result is an infinity or a QNaN, the con-
tents of the rightmost N-5 bits of the N-bit target
combination field are set to zero.

- When the result is an SNaN, bit 5 of the target for-
mats combination field is set to one and the right-
most N-6 bits of the N-bit target combination field
are set to zero.

When converting a NaN to a wider format or when con-
verting an infinity from DFP Short to DFP Long, digits in
the source trailing significand field are reencoded using
the preferred DPD codes with sufficient zeros
 appended on the left to form the target trailing signifi-
cand field. When converting a NaN to a narrower for-
mater format or when converting an infinity from DFP Long to
DFP Short, the appropriate number of leftmost digits of
the source trailing significand field are removed and the
remaining digits of the field are reencoded using the
preferred DPD codes to form the target trailing signifi-
cand field.

When converting an infinity between DFP Long and
DFP Extended, a default infinity with the same sign is
produced.

When converting an SNaN between DFP Short and
DFP Long, it is converted to an SNaN without causing
an invalid-operation exception. When converting an
SNaN between DFP Long and DFP Extended, the
invalid-operation exception occurs; if the invalid-opera-
tion exception is disabled, the result is converted to the
corresponding QNaN.

### 6.5.8.2 Data-Type Conversion

The instructions Convert From Fixed and Convert To
Fixed are provided to convert a number between the
DFP data type and the signed 64-bit binary-integer
data type.

Conversion of a signed 64-bit binary integer to a DFP
Extended number is always exact.

Conversion of a DFP number to a signed 64-bit binary
integer results in an invalid-operation exception when
the converted value does not fit into the target format,
or when the source operand is an infinity or NaN. When
the exception is disabled, the most positive integer is
returned if the source operand is a positive number or
\( +\infty \), and the most negative integer is returned if the
source operand is a negative number, \( -\infty \), or NaN.

### 6.5.9 Format Operations

The format instructions are provided to facilitate com-
posing or decomposing a DFP number, and consist of
Encode BCD To DPD, Decode DPD To BCD, Extract
Biased Exponent, Insert Biased Exponent, Shift Signifi-
cand Left Immediate, and Shift Significand Right Imme-
diate. A source operand of SNaN does not cause an
invalid-operation exception, and an SNaN may be pro-
duced as the target operand.

### 6.5.10 DFP Exceptions

This architecture defines the following DFP exceptions:

- Invalid Operation Exception
  - SNaN
  - \( \infty \pm \infty \)
  - \( 0 \div 0 \)
  - \( \infty \times 0 \)
  - Invalid Compare
  - Invalid Conversion

- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of a DFP
instruction.

Each DFP exception, and each category of the Invalid
Operation Exception, has an exception status bit in the
FPSCR. In addition, each DFP exception has a corre-
sponding enable bit in the FPSCR. The exception sta-
tus bit indicates occurrence of the corresponding
exception. If an exception occurs, the corresponding
enable bit governs the result produced by the instruc-
tion and, in conjunction with the FE0 and FE1 bits (see
the discussion of FE0 and FE1 below), whether and
how the system floating-point enabled exception error
handler is invoked. (In general, the enabling specified
by the enable bit is of invoking the system error handler,
not of permitting the exception to occur. The occur-
rence of an exception depends only on the instruction
and its source operands, not on the setting of any con-
tral bits. The only deviation from this general rule is that
the occurrence of an Underflow Exception may depend
on the setting of the enable bit.)

A single instruction, other than `mtfsfi` or `mtfsf`, may set
more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow
  Exception.
- Inexact Exception may be set with Underflow
  Exception.
- Invalid Operation Exception (SNaN) may be set
  with Invalid Operation Exception (Invalid Compare)
  for Compare Ordered instructions.
Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Conversion) for Convert To Fixed instructions.

When an exception occurs the instruction execution may be completed or partially completed, depending on the exception and the operation.

For all instructions, except for the Compare and Test instructions, the following exceptions cause the instruction execution to be partially completed. That is, setting of CR field 1 (when Rc=1) and exception status flags is performed, but no result is stored into the target FPR or FPR pair. For Compare and Test instructions, instruction execution is always completed, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exceptions, instruction execution is completed, a result, if specified by the instruction, is generated and stored into the target FPR or FPR pair, and appropriate status flags are set. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exceptions that deliver a result in target FPR are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the DFP exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, a FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case: the expectation is that the exception will be detected by software, which will revise the result. A FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to zero and Ignore Exceptions Mode (see below) should be used.

In this case the system floating-point enabled exception error handler is not invoked, even if DFP exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to one and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The system floating-point enabled exception error handler is also invoked if a Move To FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1; the Move To FPSCR instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The location of these bits and the requirements for altering them are described in Book III, Power AS Operating Environment Architecture. (The system floating-point enabled exception error handler is never invoked because of a disabled DFP exception.) The effects of the four possible settings of these bits are as follows.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions Mode</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable Mode</td>
</tr>
</tbody>
</table>

DFP exceptions do not cause the system floating-point enabled exception error handler to be invoked.

The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.

The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.
In all cases, the question of whether a DFP result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. (Recall that, for the two Imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not among those listed on page 280 as suppressed.

### Programming Note

In the ignore and both imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any exceptions, due to instructions initiated before the Floating-Point Status and Control Register instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the Floating-Point Status and Control Register instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to zero.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to one for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to one.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

#### 6.5.10.1 Invalid Operation Exception

**Definition**

An Invalid Operation Exception occurs when an operand is invalid for the specified DFP operation. The invalid DFP operations are:

- Any DFP operation on a signaling NaN (SNaN), except for Test, Round To DFP Short, Convert To DFP Long, Decode DPD To BCD, Extract Biased Exponent, Insert Biased Exponent, Shift Significand Left Immediate, and Shift Significand Right Immediate
- For add or subtract operations, magnitude subtraction of infinities \((+\infty) + (-\infty)\)
- Division of infinity by infinity \((\infty \div \infty)\)
- Division of zero by zero \((0 \div 0)\)
- Multiplication of infinity by zero \((\infty \times 0)\)
- Ordered comparison involving a NaN (Invalid Compare)
- The Quantize operation detects that the significand associated with the specified target exponent would have more significant digits than the target-format precision
- For the Quantize operation, when one source operand specifies an infinity and the other specifies a finite number
- The Reround operation detects that the target exponent associated with the specified target significance would be greater than \(X_{\text{max}}\)
- The Encode BCD To DPD operation detects an invalid BCD digit or sign code
- The Convert To Fixed operation involving a number too large in magnitude to be represented in the target format, or involving a NaN.
Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point]

6.5.10.2 Zero Divide Exception

Definition

A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value.

Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (FPSCR ZE=1) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   \[ \text{FPSCR}_{Z} = 1 \]
2. The target FPR is unchanged
3. FPSCR FR and FPRF are set to zero
4. FPSCR FPRF is set to indicate the class and sign of the result (±\(\infty\))

When Zero Divide Exception is disabled (FPSCR ZE=0) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   \[ \text{FPSCR}_{Z} = 1 \]
2. The target FPR is set to ±\(\infty\), where the sign is determined by the XOR of the signs of the operands
3. FPSCR FR and FPRF are set to zero
4. FPSCR FPRF is set to indicate the class and sign of the result (±\(\infty\))

6.5.10.3 Overflow Exception

Definition

An overflow exception occurs whenever the target format's largest finite number is exceeded in magnitude by what would have been the rounded result if the exponent range were unbounded.

Action

Except for \text{Reround}, the following describes the handling of the IEEE overflow exception condition. The \text{Reround} operation does not recognize an overflow exception condition.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.
When Overflow Exception is enabled (FPSCR\textsubscript{OE}=1) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   \( \text{FPSCR}_{OX} \leftarrow 1 \)
2. The infinitely precise result is divided by \(10^\alpha\). That is, the exponent adjustment \(\alpha\) is subtracted from the exponent. This is called the \emph{wrapped result}. The exponent adjustment for all operations, except for \emph{Round To DFP Short} and \emph{Round To DFP Long}, is 576 for DFP Long and 9216 for DFP Extended. For \emph{Round To DFP Short} and \emph{Round To DFP Long}, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the \emph{wrapped rounded result}.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. The wrapped ideal exponent is the result of subtracting the exponent adjustment from the ideal exponent.
5. FPSCR\textsubscript{FPRF} is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled (FPSCR\textsubscript{OE}=0) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   \( \text{FPSCR}_{OX} \leftarrow 1 \)
2. Inexact Exception is set
   \( \text{FPSCR}_{XX} \leftarrow 1 \)
3. The result is determined by the rounding mode and the sign of the intermediate result as follows.
4. The result is placed into the target FPR
5. FPSCR\textsubscript{FR} is set to one if the returned result is \(± \infty\), and is set to zero if the returned result is \(±N_{\text{max}}\)
6. FPSCR\textsubscript{FI} is set to one
7. FPSCR\textsubscript{FPPF} is set to indicate the class and sign of the result (± \infty or ± Normal number)

### 6.5.10.4 Underflow Exception

#### Definition

Except for \emph{Reround}, the following describes the handling of the IEEE underflow exception condition. The \emph{Reround} operation does not recognize an underflow exception condition.

The Underflow Exception is defined differently for the enabled and disabled states. However, a tininess condition is recognized in both states when a result computed as though both the precision and exponent range were unbounded would be nonzero and less than the target format’s smallest normal number, \(N_{\text{min}}\), in magnitude.

Unless otherwise defined in the instruction description, an underflow exception occurs as follows:

- **Enabled:**
  - When the tininess condition is recognized.
- **Disabled:**
  - When the tininess condition is recognized and when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

#### Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled (FPSCR\textsubscript{UE}=1) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   \( \text{FPSCR}_{UX} \leftarrow 1 \)
2. The infinitely precise result is multiplied by \(10^\alpha\). That is, the exponent adjustment \(\alpha\) is added to the exponent. This is called the \emph{wrapped result}. The exponent adjustment for all operations, except for \emph{Round To DFP Short} and \emph{Round To DFP Long}, is 576 for DFP Long and 9216 for DFP Extended. For \emph{Round To DFP Short} and \emph{Round To DFP Long}, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the \emph{wrapped rounded result}.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Sign of intermediate result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus</td>
<td>Minus</td>
</tr>
<tr>
<td>Round to Nearest, Ties to Even</td>
<td>+∞</td>
</tr>
<tr>
<td>Round toward 0</td>
<td>+N_{\text{max}}</td>
</tr>
<tr>
<td>Round toward +∞</td>
<td>+∞</td>
</tr>
<tr>
<td>Round toward −∞</td>
<td>+N_{\text{max}}</td>
</tr>
<tr>
<td>Round to Nearest, Ties away from 0</td>
<td>+∞</td>
</tr>
<tr>
<td>Round to Nearest, Ties toward 0</td>
<td>+∞</td>
</tr>
<tr>
<td>Round away from 0</td>
<td>+∞</td>
</tr>
<tr>
<td>Round to prepare for shorter precision</td>
<td>+N_{\text{max}}</td>
</tr>
</tbody>
</table>

Figure 88. Overflow Results When Exception Is Disabled
wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of adding the exponent adjustment to the ideal exponent.

5. FPSCR<sub>FPRF</sub> is set to indicate the class and sign of the result (± Normal number)

When Underflow Exception is disabled (FPSCR<sub>UE</sub>=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   \[ \text{FPSCR}_{UX} \leftarrow 1 \]
2. The infinitely precise result is rounded to the target-format precision.
3. The rounded result is returned. If this result has redundant forms, the result of the form that is closest to the ideal exponent is returned.
4. FPSCR<sub>FPRF</sub> is set to indicate the class and sign of the result (± Normal number, ± Subnormal Number, or ± Zero)

### 6.5.10.5 Inexact Exception

**Definition**

Except for *Round to FP Integer Without Inexact*, the following describes the handling of the IEEE inexact exception condition. The *Round to FP Integer Without Inexact* does not recognize an inexact exception condition.

An Inexact Exception occurs when either of two conditions occur during rounding:

1. The delivered result differs from what would have been computed were both the precision and exponent range unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

**Action**

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   \[ \text{FPSCR}_{WX} \leftarrow 1 \]
2. The rounded or overflowed result is placed into the target FPR
3. FPSCR<sub>FPRF</sub> is set to indicate the class and sign of the result

**Programming Note**

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.
6.5.11 Summary of Normal Rounding And Range Actions

Figure 89 and Figure 90 summarize rounding and range actions, with the following exceptions:

- The Reround operation recognizes neither an underflow nor an overflow exception.
- The Round to FP Integer Without Inexact operation does not recognize the inexact operation exception.

<table>
<thead>
<tr>
<th>Range of v</th>
<th>Case</th>
<th>Result (r) when Rounding Mode Is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RNE</td>
</tr>
<tr>
<td>v &lt; -Nmax, q &lt; -Nmax</td>
<td>Overflow</td>
<td>+∞</td>
</tr>
<tr>
<td>v &lt; -Nmax, q = -Nmax</td>
<td>Normal</td>
<td>-Nmax</td>
</tr>
<tr>
<td>-Nmax ≤ v ≤ -Nmin</td>
<td>Normal</td>
<td>b</td>
</tr>
<tr>
<td>-Nmin &lt; v ≤ -Dmin</td>
<td>Tiny</td>
<td>b*</td>
</tr>
<tr>
<td>-Dmin/2 &lt; v &lt; 0</td>
<td>Tiny</td>
<td>0</td>
</tr>
<tr>
<td>v = 0</td>
<td>EZD</td>
<td>0</td>
</tr>
<tr>
<td>+Dmin/2/2 &lt; v &lt; +Dmin/2</td>
<td>Tiny</td>
<td>+Dmin</td>
</tr>
<tr>
<td>+Dmin ≤ v &lt; +Nmin</td>
<td>Tiny</td>
<td>b*</td>
</tr>
<tr>
<td>+Nmin ≤ v ≤ +Nmax</td>
<td>Normal</td>
<td>b</td>
</tr>
<tr>
<td>+Nmax &lt; v, q = +Nmax</td>
<td>Normal</td>
<td>+Nmax</td>
</tr>
<tr>
<td>+Nmax &lt; v, q &gt; +Nmax</td>
<td>Overflow</td>
<td>+∞</td>
</tr>
</tbody>
</table>

Explanation:

- This situation cannot occur.
- The normal result r is considered to have been incremented.
- The rounded value, in the extreme case, may be Nmin. In this case, the exception conditions are underflow, inexact, and incremented.
- The value derived when the precise result v is rounded to the destination's precision, including both bounded precision and bounded exponent range.
- The value derived when the precise result v is rounded to the destination's precision, but assuming an unbounded exponent range.
- This is the returned value when neither overflow nor underflow is enabled.
- Precise result before rounding, assuming unbounded precision and an unbounded exponent range. For data-format conversion operations, v is the source value.
- Smallest (in magnitude) representable subnormal number in the target format.
- The result r of the exact-zero-difference case applies only to ADD and SUBTRACT with both source operands having opposite signs. (For ADD and SUBTRACT, when both source operands have the same sign, the sign of the zero result is the same sign as the sign of the source operands.)
- Largest (in magnitude) representable finite number in the target format.
- Smallest (in magnitude) representable normalized number in the target format.
- Round away from 0.
- Round to Prepare for Shorter Precision.
- Round to Nearest, Ties away from 0.
- Round to Nearest, Ties to even.
- Round toward +∞.
- Round toward −∞.
- Round toward 0.

Figure 89. Rounding and Range Actions (Part 1)
| Case       | Is r inexact (r≠v) | OE=1 | UE=1 | XE=1 | Is r incremented (|r|>|v|) | Is q inexact (q≠v) | Is q incremented (|q|>|v|) | Returned Results and Status Setting* |
|------------|--------------------|------|------|------|-------------------------------|--------------------|----------------------|-----------------------------------|
| Overflow   | Yes                | No   | —    | No   | —                            | —                  | —                    | —                                 |
| Overflow   | Yes                | No   | —    | Yes  | —                            | —                  | —                    | —                                 |
| Overflow   | Yes                | No   | Yes  | No   | —                            | —                  | —                    | —                                 |
| Overflow   | Yes                | No   | Yes  | Yes  | —                            | —                  | —                    | —                                 |
| Overflow   | Yes                | Yes  | —    | —    | No                           | No                 | No                   | Ox                               |
| Overflow   | Yes                | Yes  | —    | —    | Yes                          | No                 | No                   | OX                               |
| Overflow   | Yes                | Yes  | —    | —    | Yes                          | No                 | No                   | OX                               |
| Overflow   | Yes                | No   | Yes  | Yes  | —                            | —                  | —                    | —                                 |
| Overflow   | Yes                | Yes  | —    | —    | Yes                          | Yes               | Yes                  | Ox                               |
| Overflow   | Yes                | Yes  | —    | —    | Yes                          | Yes               | Yes                  | Ox                               |
| Overflow   | Yes                | Yes  | —    | —    | Yes                          | Yes               | Yes                  | Ox                               |
| Normal     | No                 | —    | —    | No   | —                            | —                  | —                    | —                                 |
| Normal     | Yes                | —    | —    | No   | —                            | —                  | —                    | —                                 |
| Normal     | Yes                | —    | No   | Yes  | —                            | —                  | —                    | —                                 |
| Normal     | Yes                | —    | Yes  | No   | —                            | —                  | —                    | —                                 |
| Normal     | Yes                | —    | Yes  | Yes  | —                            | —                  | —                    | —                                 |
| Tiny       | No                 | —    | —    | No   | —                            | —                  | —                    | —                                 |
| Tiny       | Yes                | —    | —    | —    | No                           | No                 | No                   | Ox                               |
| Tiny       | Yes                | —    | —    | —    | Yes                          | —                  | —                    | Ox                               |
| Tiny       | Yes                | —    | —    | —    | Yes                          | —                  | —                    | Ox                               |
| Tiny       | Yes                | —    | —    | —    | Yes                          | —                  | —                    | Ox                               |
| Tiny       | Yes                | —    | —    | —    | Yes                          | —                  | —                    | Ox                               |

Explained:
— The results do not depend on this condition.
1 This condition is true by virtue of the state of some condition to the left of this column.
Round sets only the FI and FR status flags. Setting of the OX, XX, or UX flag is part of the exception actions. They are listed here for reference.

\( \beta \) Wrap adjust, which depends on the type of operation and operand format. For all operations except Round to DFP Short and Round to DFP Long, the wrap adjust depends on the target format: \( \beta = 10^\kappa \), where \( \kappa \) is 576 for DFP Long, and 9216 for DFP Extended. For Round to DFP Short and Round to DFP Long, the wrap adjust depends on the source format: \( \beta = 10^\kappa \) where \( \kappa \) is 192 for DFP Long and 3072 for DFP Extended.

\( q \) The value derived when the precise result \( v \) is rounded to destination's precision, but assuming an unbounded exponent range.

\( r \) The result as defined in Part 1 of this figure.

\( \nu \) Precise result before rounding, assuming unbounded precision and unbounded exponent range.

FI Floating-Point-Fraction-Inexact status flag, FPSCR\(_{FI}\). This status flag is non-sticky.

FR Floating-Point-Fraction-Rounded status flag, FPSCR\(_{FR}\).

OX Floating-Point Overflow Exception status flag, FPSCR\(_{UX}\).

TU The system floating-point enabled exception error handler is invoked for the overflow exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.

TX The system floating-point enabled exception error handler is invoked for the underflow exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.

T(x) The value \( x \) is placed at the target operand location.

Tw(x) The wrapped rounded result \( x \) is placed at the target operand location. For all operations except data format conversions, the wrapped rounded result is in the same format and length as normal results at the target location. For data format conversions, the wrapped rounded result is in the same format and length as the source, but rounded to the target-format precision.

UX Floating-Point-Underflow-Exception status flag, FPSCR\(_{UX}\).

XX Float-Point-Inexact-Exception Status flag, FPSCR\(_{XX}\). The flag is a sticky version of FPSCR\(_{FI}\). When FPSCR\(_{FI}\) is set to a new value, the new value of FPSCR\(_{XX}\) is set to the result of ORing the old value of FPSCR\(_{XX}\) with the new value of FPSCR\(_{FI}\).

Figure 90. Rounding and Range Actions (Part 2)
6.6 DFP Instruction Descriptions

The following sections describe the DFP instructions. When a 128-bit operand is used, it is held in a FPR pair and the instruction mnemonic uses a letter “q” to mean the quad-precision operation. Note that in the following descriptions, FPXp denotes a FPR pair and must address an even-odd pair. If the FPXp field specifies an odd-numbered register, then the instruction form is invalid. The notation FPX[p] means either a FPR, FPX, or a FPR pair, FPXp.

For DFP instructions, if a DFP operand is returned, the trailing significand field of the target operand is encoded using preferred DPD codes.
6.6.1 DFP Arithmetic Instructions

All DFP arithmetic instructions are X-form instructions. They all set the FI and FR status flags, and also set the FPSCR\textsubscript{FPRF} field. Furthermore, they all have an ideal exponent assigned and employ the record bit (Rc).

The arithmetic instructions consist of Add, Divide, Multiply, and Subtract.

### DFP Add [Quad] X-form

- dadd: FRT, FRA, FRB (Rc=0)
- dadd: FRT, FRA, FRB (Rc=1)
- daddq: FRTp, FRAp, FRBp (Rc=0)
- daddq: FRTp, FRAp, FRBp (Rc=1)

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>FRB</th>
<th>2</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The DFP operand in FRB[p] is added to the DFP operand in FRA[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

Figure 91 summarizes the actions for Add. Figure 91 does not include the setting of the FPSCR\textsubscript{FPRF} field. The FPSCR\textsubscript{FPRF} field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

**Special Registers Altered:**
- FPRF
- FR
- FI
- FX
- OX
- UX
- XX
- VXSNAN
- VXISI
- CR1 (if Rc=1)

### DFP Subtract [Quad] X-form

- dsub: FRT, FRA, FRB (Rc=0)
- dsub: FRT, FRA, FRB (Rc=1)
- dsubq: FRTp, FRAp, FRBp (Rc=0)
- dsubq: FRTp, FRAp, FRBp (Rc=1)

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>FRB</th>
<th>514</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The DFP operand in FRB[p] is subtracted from the DFP operand in FRA[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

The execution of Subtract is identical to that of Add, except that the operand in FRB participates in the operation with its sign bit inverted. See Figure 91. The table does not include the setting of the FPSCR\textsubscript{FPRF} field. The FPSCR\textsubscript{FPRF} field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

**Special Registers Altered:**
- FPRF
- FR
- FI
- FX
- OX
- UX
- XX
- VXSNAN
- VXISI
- CR1 (if Rc=1)
## Figure 91. Actions: Add

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>$-\infty$</th>
<th>F</th>
<th>$+\infty$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>T($-dINF$)</td>
<td>T($-dINF$)</td>
<td>V\text{XISI}: T(dNaN)</td>
<td>P(b)</td>
<td>V\text{XSNAN}: U(b)</td>
</tr>
<tr>
<td>F</td>
<td>T($-dINF$)</td>
<td>S(a + b)</td>
<td>T($+dINF$)</td>
<td>P(b)</td>
<td>V\text{XSNAN}: U(b)</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>V\text{XISI}: T(dNaN)</td>
<td>T($+dINF$)</td>
<td>T($+dINF$)</td>
<td>P(b)</td>
<td>V\text{XSNAN}: U(b)</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>V\text{XSNAN}: U(b)</td>
</tr>
<tr>
<td>SNaN</td>
<td>V\text{XSNAN}: U(a)</td>
<td>V\text{XSNAN}: U(a)</td>
<td>V\text{XSNAN}: U(a)</td>
<td>V\text{XSNAN}: U(a)</td>
<td>V\text{XSNAN}: U(a)</td>
</tr>
</tbody>
</table>

**Explanation:**

- $a + b$ The value $a$ added to $b$, rounded to the target-format precision and returned in the appropriate form. (See Section 6.5.11 on page 286)
- $+dINF$ Default plus infinity.
- $-dINF$ Default minus infinity.
- $dNaN$ Default quiet NaN.
- F All finite numbers, including zeros.
- P(x) The QNaN of operand $x$ is propagated and placed in FRT[p].
- S(x) The value $x$ is placed in FRT[p] with the sign set by the rules of algebra. When the source operands have the same sign, the sign of the result is the same as the sign of the operands, including the case when the result is zero. When the operands have opposite signs, the sign of a zero result is positive in all rounding modes, except round toward $-\infty$, in which case, the sign is minus.
- T(x) The value $x$ is placed in FRT[p].
- U(x) The SNaN of operand $x$ is converted to the corresponding QNaN and placed in FRT[p].
- V\text{XISI} The Invalid-Operation Exception (VXISI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
- V\text{XSNAN} The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
**DFP Multiply [Quad]**

<table>
<thead>
<tr>
<th>DFP Multiply</th>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmul</td>
<td>FRT, FRA, FRB (Rc=0)</td>
</tr>
<tr>
<td>dmul.</td>
<td>FRT, FRA, FRB (Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special Registers Altered:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRF FR FI</td>
</tr>
<tr>
<td>FX OX UX XX</td>
</tr>
<tr>
<td>VXSNAN VXIMZ</td>
</tr>
<tr>
<td>CR1</td>
</tr>
</tbody>
</table>

The DFP operand in FRA[p] is multiplied by the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the sum of the two exponents of the source operands.

Figure 92 summarizes the actions for Multiply.

Figure 92 does not include the setting of the FPSCR\_FPRF field. The FPSCR\_FPRF field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Multiply (a*b) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S(a * b)</td>
</tr>
<tr>
<td>Fn</td>
<td>S(a * b)</td>
</tr>
<tr>
<td>∞</td>
<td>V_XIMZ_T(dNaN)</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
</tr>
<tr>
<td>SNaN</td>
<td>V_XSNAN_U(a)</td>
</tr>
<tr>
<td>dINF</td>
<td>Default infinity.</td>
</tr>
<tr>
<td>dNaN</td>
<td>Default quiet NaN.</td>
</tr>
<tr>
<td>Fn</td>
<td>Finite nonzero number (includes both normal and subnormal numbers).</td>
</tr>
<tr>
<td>P(x)</td>
<td>The QNaN of operand x is propagated and placed in FRT[p].</td>
</tr>
<tr>
<td>S(x)</td>
<td>The value x is placed in FRT[p] with the sign set to the exclusive-OR of the source-operand signs.</td>
</tr>
<tr>
<td>T(x)</td>
<td>The value x is placed in FRT[p].</td>
</tr>
<tr>
<td>U(x)</td>
<td>The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].</td>
</tr>
<tr>
<td>V_XIMZ_T</td>
<td>The Invalid-Operation Exception (VXIMZ) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)</td>
</tr>
<tr>
<td>V_XSNAN_U</td>
<td>The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)</td>
</tr>
</tbody>
</table>

**Figure 92. Actions: Multiply**
**DFP Divide [Quad] X-form**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ddiv</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddiv.</td>
<td>FRT,FRA,FRB</td>
<td>(Rc=1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>59</td>
<td>FRT</td>
<td>FRA</td>
<td>FRB</td>
</tr>
<tr>
<td>ddivq</td>
<td>FRTp,FRAp,FRBp</td>
<td>(Rc=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddivq.</td>
<td>FRTp,FRAp,FRBp</td>
<td>(Rc=1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>63</td>
<td>FRTp</td>
<td>FRAp</td>
<td>FRBp</td>
</tr>
</tbody>
</table>

The DFP operand in FRA[p] is divided by the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the difference of subtracting the exponent of the divisor from the exponent of the dividend.

**Figure 93 summarizes the actions for Divide.** Figure 93 does not include the setting of the FPSCR FPRF field. The FPSCR FPRF field is always set to the class and sign of the result, except for an enabled invalid-operation and enabled zero-divide exceptions, in which cases the field remains unchanged.

**Special Registers Altered:**
- FPRF FR FI
- FX OX UX ZK XX
- VXSNAN VXIDI VXZDZ CR1

**Operand a in FRA[p] is**

| Actions for Divide (a ÷ b) when operand b in FRB[p] is |
|------------|----------|----------|----------|----------|
| 0 | Fn | ∞ | QNaN | SNaN |

- *VXZD*: T(dNaN) S(a ÷ b) S(zt) P(b) VXSNAN: U(b)
- *VXID*: T(dNaN) S(a ÷ b) S(zt) P(b) VXSNAN: U(b)
- *VXIDI*: P(a) P(a) P(a) P(a) VXSNAN: U(b)
- *VXSNAN*: V(a) V(a) V(a) V(a) V(a)

**Explanation:**
- a ÷ b The value a divided by b, rounded to the target-format precision and returned in the appropriate form. (See Section 6.5.11 on page 286.)
- dINF Default infinity.
- dNaN Default quiet NaN.
- Fn Finite nonzero number (includes both normal and subnormal numbers).
- P(x) The QNaN of operand x is propagated and placed in FRT[p].
- S(x) The value x is placed in FRT[p] with the sign set to the exclusive-OR of the source-operand signs.
- T(x) The value x is placed in FRT[p].
- U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- VXIDI: The Invalid-Operation Exception (VXIDI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
- VXSNAN: The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
- VXZD: The Invalid-Operation Exception (VXZD) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
- zt True zero (zero significand and most negative exponent).
- Zx The Zero-Divide Exception occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.2 “Zero Divide Exception” on page 283 for the exception actions.)
6.6.2 DFP Compare Instructions

The DFP compare instructions consist of the *Compare Ordered* and *Compare Unordered* instructions. The compare instructions do not provide the record bit.

The comparison sets the designated CR field to indicate the result. The FPSCR\textsubscript{FPCC} is set in the same way.

The codes in the CR field BF and FPSCR\textsubscript{FPCC} are defined for the DFP compare operations as follows:

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FL (FRA[p]) &lt; (FRB[p])</td>
</tr>
<tr>
<td>1</td>
<td>FG (FRA[p]) &gt; (FRB[p])</td>
</tr>
<tr>
<td>2</td>
<td>FE (FRA[p]) = (FRB[p])</td>
</tr>
<tr>
<td>3</td>
<td>FU (FRA[p]) ? (FRB[p])</td>
</tr>
</tbody>
</table>
DFP Compare Unordered [Quad]  X-form

dcmpu  BF, FRA, FRB

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
& BF & FRA & FRB & 642 & \quad /
\hline
0 & 59 & 9 & 11 & 16 & 21 & 31
\hline
\end{array}
\]

dcmpuq  BF, FRAp, FRBp

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
& BF & FRAp & FRBp & 642 & \quad /
\hline
0 & 63 & 9 & 11 & 16 & 21 & 31
\hline
\end{array}
\]

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPSCRFPCC.

**Special Registers Altered:**
- CR field BF
- FPCC
- FX
- VXSNAN

### Table: Actions for Compare Unordered (a:b) when operand b in FRB[p] is

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Compare Unordered (a:b) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>AeqB, AltB, AltB, AuoB, Fu, VXSNAN</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>AeqB, AltB, AuoB</td>
</tr>
<tr>
<td>F</td>
<td>AltB, AeqB, Fu, VXSNAN</td>
</tr>
<tr>
<td>NaN</td>
<td>TaqB, AuoB</td>
</tr>
<tr>
<td>NaN</td>
<td>Fu, VXSNAN</td>
</tr>
</tbody>
</table>

**Explanation:**
- \(C(a:b)\)  Algebraic comparison. See the table below.
- F          All finite numbers, including zeros.
- AeqB       CR field BF and FPSCRFPCC are set to 0b0010.
- AgtB       CR field BF and FPSCRFPCC are set to 0b0100.
- AltB       CR field BF and FPSCRFPCC are set to 0b1000.
- AuoB       CR field BF and FPSCRFPCC are set to 0b0001.
- VXSNAN     The invalid-operation exception (VXSNAN) occurs. See Section 6.5.10.1 for actions.

### Table: Relation of Value a to Value b

<table>
<thead>
<tr>
<th>Relation of Value a to Value b</th>
<th>Action for (C(a:b))</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>AeqB</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>AltB</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>AgtB</td>
</tr>
</tbody>
</table>

Figure 94. Actions: Compare Unordered
DFP Compare Ordered [Quad]  

X-form

dcmpo  BF,FRA,FRB

<table>
<thead>
<tr>
<th>59</th>
<th>BF // FRA</th>
<th>FRB</th>
<th>130</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>9</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

dcmpoq  BF,FRAp,FRBp

<table>
<thead>
<tr>
<th>63</th>
<th>BF // FRAp</th>
<th>FRBp</th>
<th>130</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>9</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPSCRFPCC.

Special Registers Altered:
- CR field BF
- FPCC
- FX
- VXSNAN
- VXVC

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the FPSCRFPCC.

Special Registers Altered:
- CR field BF
- FPCC
- FX
- VXSNAN
- VXVC

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Compare ordered (a:b) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>AeQB</td>
</tr>
<tr>
<td>F</td>
<td>AgtB</td>
</tr>
<tr>
<td>+∞</td>
<td>AgtB</td>
</tr>
<tr>
<td>QNaN</td>
<td>AuoB, VXVC</td>
</tr>
<tr>
<td>SNaN</td>
<td>AuoB, VXSV</td>
</tr>
</tbody>
</table>

Explanation:
- C(a:b)  Algebraic comparison. See the table below
- F     All finite numbers, including zeros
- AeQB  CR field BF and FPSCRFPCC are set to 0b0010.
- AgtB  CR field BF and FPSCRFPCC are set to 0b0100.
- AltB  CR field BF and FPSCRFPCC are set to 0b1000.
- AuoB  CR field BF and FPSCRFPCC are set to 0b0001.
- VXSV  The invalid-operation exception (VXSNAN) occurs. Additionally, if the exception is disabled (FPSCRVE=0), then FPSCRVXVC is also set to one. See Section 6.5.10.1 for actions.
- VXVC  The invalid-operation exception (VXVC) occurs. See Section 6.5.10.1 for actions.

<table>
<thead>
<tr>
<th>Relation of Value a to Value b</th>
<th>Action for C(a:b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>AeQB</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>AltB</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>AgtB</td>
</tr>
</tbody>
</table>

Figure 95. Actions: Compare Ordered
6.6.3 DFP Test Instructions

The DFP test instructions consist of the Test Data Class, Test Data Group, Test Exponent, and Test Significance instructions, and they do not provide the record bit. The test instructions set the designated CR field to indicate the result. The FPSCR_{FPCC} is set in the same way.

**DFP Test Data Class [Quad]**

<table>
<thead>
<tr>
<th>DCM Bit</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>1</td>
<td>Subnormal</td>
</tr>
<tr>
<td>2</td>
<td>Normal</td>
</tr>
<tr>
<td>3</td>
<td>Infinity</td>
</tr>
<tr>
<td>4</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>5</td>
<td>Signaling NaN</td>
</tr>
</tbody>
</table>

**Field**

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Operand positive with no match</td>
</tr>
<tr>
<td>0010</td>
<td>Operand positive with match</td>
</tr>
<tr>
<td>1000</td>
<td>Operand negative with no match</td>
</tr>
<tr>
<td>1010</td>
<td>Operand negative with match</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**
- CR field BF
- FPCC

Let the DCM (Data Class Mask) field specify one or more of the 6 possible data classes, where each bit corresponds to a specific data class.

Let the DGM (Data Group Mask) field specify one or more of the 6 possible data groups, where each bit corresponds to a specific data group.

The term extreme exponent means either the maximum exponent, \(X_{\text{max}}\), or the minimum exponent, \(X_{\text{min}}\).

**DFP Test Data Group [Quad]**

<table>
<thead>
<tr>
<th>DGM Bit</th>
<th>Data Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero with non-extreme exponent</td>
</tr>
<tr>
<td>1</td>
<td>Zero with extreme exponent</td>
</tr>
<tr>
<td>2</td>
<td>Subnormal or (Normal with extreme exponent)</td>
</tr>
<tr>
<td>3</td>
<td>Normal with non-extreme exponent and leftmost zero digit in significand</td>
</tr>
<tr>
<td>4</td>
<td>Normal with non-extreme exponent and leftmost nonzero digit in significand</td>
</tr>
<tr>
<td>5</td>
<td>Special symbol (Infinity, QNaN, or SNaN)</td>
</tr>
</tbody>
</table>

**Field**

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Operand positive with no match</td>
</tr>
<tr>
<td>0010</td>
<td>Operand positive with match</td>
</tr>
<tr>
<td>1000</td>
<td>Operand negative with no match</td>
</tr>
<tr>
<td>1010</td>
<td>Operand negative with match</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**
- CR field BF
- FPCC

**DFP Test Data Class [Quad] Z22-form**

<table>
<thead>
<tr>
<th>59 BF // FRA DCM</th>
<th>194</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 9 11 16 22 31</td>
</tr>
</tbody>
</table>

dtstdc BF,FRA,DCM

<table>
<thead>
<tr>
<th>63 BF // FRAp DCM</th>
<th>194</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 9 11 16 22 31</td>
</tr>
</tbody>
</table>

dtstdcq BF,FRAp,DCM

**DFP Test Data Group [Quad] Z22-form**

<table>
<thead>
<tr>
<th>59 BF // FRA DGM</th>
<th>226</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 9 11 16 22 31</td>
</tr>
</tbody>
</table>

dtstdg BF,FRA,DGM

<table>
<thead>
<tr>
<th>63 BF // FRAp DGM</th>
<th>226</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 9 11 16 22 31</td>
</tr>
</tbody>
</table>

dtstdgq BF,FRAp,DGM
**DFP Test Exponent [Quad] X-form**

dstex BF,FRA,FRB

<table>
<thead>
<tr>
<th>59</th>
<th>BF</th>
<th>FRA</th>
<th>FRB</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

dstexq BF,FRAp,FRBp

<table>
<thead>
<tr>
<th>63</th>
<th>BF</th>
<th>FRAp</th>
<th>FRBp</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

The exponent value (Ea) of the DFP operand in FRA[p] is compared to the exponent value (Eb) of the DFP operand in FRB [p]. The result of the compare is placed into CR field BF and the FPSCRFPCC.

The codes in the CR field BF and FPSCRFPCC are defined for the DFP Test Exponent operations as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ea &lt; Eb</td>
</tr>
<tr>
<td>1</td>
<td>Ea &gt; Eb</td>
</tr>
<tr>
<td>2</td>
<td>Ea = Eb</td>
</tr>
<tr>
<td>3</td>
<td>Ea ? Eb</td>
</tr>
</tbody>
</table>

**Special Registers Altered:**
- CR field BF
- FPCC

**Operand a in FRA[p] is**

<table>
<thead>
<tr>
<th>Actions for Test Exponent (Ea:Eb) when operand b in FRB[p] is</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>∞</td>
</tr>
<tr>
<td>QNaN</td>
</tr>
<tr>
<td>SNaN</td>
</tr>
</tbody>
</table>

**Explanation:**
- C(Ea:Eb) Algebraic comparison. See the table below.
- F All finite numbers, including zeros
- AeqB CR field BF and FPSCRFPCC are set to 0b0010.
- AgtB CR field BF and FPSCRFPCC are set to 0b0010.
- AltB CR field BF and FPSCRFPCC are set to 0b0100.
- AuoB CR field BF and FPSCRFPCC are set to 0b0001.

**Relation of Value Ea to Value Eb**

<table>
<thead>
<tr>
<th>Action for C(Ea:Eb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ea = Eb</td>
</tr>
<tr>
<td>Ea &lt; Eb</td>
</tr>
<tr>
<td>Ea &gt; Eb</td>
</tr>
</tbody>
</table>

Figure 96. Actions: Test Exponent
DFP Test Significance [Quad]  X-form

dtstsf  BF,FRA,FRB

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>k ≠ 0 and k &lt; NSDb</td>
</tr>
<tr>
<td>1</td>
<td>k ≠ 0 and k &gt; NSDb, or k = 0</td>
</tr>
<tr>
<td>2</td>
<td>k ≠ 0 and k = NSDb</td>
</tr>
<tr>
<td>3</td>
<td>k ? NSDb</td>
</tr>
</tbody>
</table>

Special Registers Altered:
CR field BF
FPCC

Programming Note
The reference significance can be loaded into a FPR using a Load Float as Integer Word Algebraic instruction

dtstsfq  BF,FRA,FRBp

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Actions for Test Significance when the operand in FRB[p] is

| C(k: NSDb) | AeqB | AuoB | AuoB |
| F   | ∞    | QNaN | SNaN |

Explanation:
C(k: NSDb) Algebraic comparison. See the table below.
F All finite numbers, including zeros.
AeqB CR field BF and FPSCR_FPCC are set to 0b0010.
AgtB CR field BF and FPSCR_FPCC are set to 0b0100.
AltB CR field BF and FPSCR_FPCC are set to 0b1000.
AuoB CR field BF and FPSCR_FPCC are set to 0b0001.

Relation of Value NSDb to Value k

<table>
<thead>
<tr>
<th>Action for C(k:NSDb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k ≠ 0 and k = NSDb</td>
</tr>
<tr>
<td>k ≠ 0 and k &lt; NSDb</td>
</tr>
<tr>
<td>k ≠ 0 and k &gt; NSDb, or k = 0</td>
</tr>
</tbody>
</table>

Figure 97. Actions: Test Significance
6.6.4 DFP Quantum Adjustment Instructions

The Quantum Adjustment operations consist of the Quantize, Quantize Immediate, Reround, and Round To FP Integer operations.

The Quantum Adjustment instructions are Z23-form instructions and have an immediate RMC (Rounding-Mode-Control) field, which specifies the rounding mode used. For Quantize, Quantize Immediate, and Reround, the RMC field contains the primary encoding. For Round To FP Integer, the field contains either primary or secondary encoding, depending on the setting of a RMC-encoding-selection bit. See Section 6.5.2 “Rounding Mode Specification” on page 277 for the definition of RMC encoding.

All Quantum Adjustment instructions set the FI and FR status flags, and also set the FPSCR_FPRF field. The record bit is provided to each of these instructions. They return the target operand in a form with the ideal exponent.

DFP Quantize Immediate [Quad] Z23-form

\[
\begin{array}{cccccc}
59 & FRT & TE & FRB & RMC & 67 \\
0 & 6 & 11 & 16 & 21 & 23 \\
\end{array}
\]

\[
\begin{array}{cccccc}
63 & FRTp & TE & FRBp & RMC & 67 \\
0 & 6 & 11 & 16 & 21 & 23 \\
\end{array}
\]

The DFP operand in FRB[p] is converted and rounded to the form with the exponent specified by TE based on the rounding mode specified in the RMC field. TE is a 5-bit signed binary integer. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified by TE.

When the value of the operand in FRB[p] is greater than \((10^p-1) \times 10^{TE}\), where \(p\) is the format precision, an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

The FPSCR_FPRF field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

- FPRF
- FR
- FI
- FX
- VXSNAN
- VXCVI
- CR1 (if Rc=1)

Programming Note

DFP Quantize Immediate can be used to adjust values to a form having the specified exponent in the range -16 to 15. If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN.;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field.

DFP Quantize Immediate can round a value to a specific number of fractional digits. Consider the computation of sales tax. Values expressed in U.S. dollars have 2 fractional digits, and sales tax rates typically have 3 fractional digits. The product of value and rate will yield 5 fractional digits. For example:

\[
39.95 \times 0.075 = 2.99625
\]

This result needs to be rounded to the penny to compute the correct tax of $3.00.

The following sequence computes the sales tax assuming the pre-tax total is in FRA and the tax rate is in FRB. The DFP Quantize Immediate instruction rounds the product (FRA * FRB) to 2 fractional digits (TE field = -2) using Round to nearest, ties away from 0 (RMC field = 2). The quantized and rounded result is placed in FRT.

\[
dmul \\ dquai -2,FRT,f0,2
\]
**DFP Quantize [Quad]**

The DFP operand in register FRB[p] is converted and rounded to the form with the same exponent as that of the DFP operand in FRA[p] based on the rounding mode specified in the RMC field. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified in FRA[p].

When the value of the operand in FRB[p] is greater than \((10^p-1) \times 10^{E_a}\), where \(p\) is the format precision and \(E_a\) is the exponent of the operand in FRA[p], an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 99 and Figure 100 summarize the actions. The tables do not include the setting of the FPSCR\textsubscript{FRF} field. The FPSCR\textsubscript{FRF} field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

---

**Special Register Altered:**

<table>
<thead>
<tr>
<th></th>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FX</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>VXSNAN</td>
<td>VXCVI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**(if Rc=1)**

---

**Programming Note**

**DFP Quantize** can be used to adjust one DFP value (FRB[p]) to a form having the same exponent as a second DFP value (FRA[p]). If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field. Figure 98 shows examples of these adjustments.

---

**Table:**

<table>
<thead>
<tr>
<th>FRA</th>
<th>FRB</th>
<th>FRT when RMC=1</th>
<th>FRT when RMC=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ((1 \times 10^0))</td>
<td>9. ((9 \times 10^0))</td>
<td>9 ((9 \times 10^0))</td>
<td>9 ((9 \times 10^0))</td>
</tr>
<tr>
<td>1.00 ((100 \times 10^{-2}))</td>
<td>9. ((9 \times 10^0))</td>
<td>9.00 ((900 \times 10^{-2}))</td>
<td>9.00 ((900 \times 10^{-2}))</td>
</tr>
<tr>
<td>1 ((1 \times 10^0))</td>
<td>49.1234 ((491234 \times 10^{-4}))</td>
<td>49 ((49 \times 10^0))</td>
<td>49 ((49 \times 10^0))</td>
</tr>
<tr>
<td>1.00 ((100 \times 10^{-2}))</td>
<td>49.1234 ((491234 \times 10^{-4}))</td>
<td>49.12 ((4912 \times 10^{-2}))</td>
<td>49.12 ((4912 \times 10^{-2}))</td>
</tr>
<tr>
<td>1 ((1 \times 10^0))</td>
<td>49.9876 ((499876 \times 10^{-4}))</td>
<td>49 ((49 \times 10^0))</td>
<td>50 ((50 \times 10^0))</td>
</tr>
<tr>
<td>1.00 ((100 \times 10^{-2}))</td>
<td>49.9876 ((499876 \times 10^{-4}))</td>
<td>49.98 ((4998 \times 10^{-2}))</td>
<td>49.99 ((4999 \times 10^{-2}))</td>
</tr>
<tr>
<td>0.01 ((1 \times 10^{-2}))</td>
<td>49.9876 ((499876 \times 10^{-4}))</td>
<td>49.98 ((4998 \times 10^{-2}))</td>
<td>49.99 ((4999 \times 10^{-2}))</td>
</tr>
<tr>
<td>1 ((1 \times 10^0))</td>
<td>9999999999999999 ((9999999999999999 \times 10^0))</td>
<td>9999999999999999 ((9999999999999999 \times 10^0))</td>
<td>9999999999999999 ((9999999999999999 \times 10^0))</td>
</tr>
<tr>
<td>1.0 ((10 \times 10^{-1}))</td>
<td>9999999999999999 ((9999999999999999 \times 10^0))</td>
<td>QNaN</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

**Figure 98. DFP Quantize examples**
### Figure 99. Actions (part 1) Quantize

<table>
<thead>
<tr>
<th>Operand a in FRA[p] is</th>
<th>Actions for Quantize when operand b in FRB[p] is</th>
<th>0</th>
<th>Fn</th>
<th>∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>V_SXCVI: T(dNaN)</td>
<td>P(b)</td>
</tr>
<tr>
<td>Fn</td>
<td>•</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>V_SXCVI: T(dNaN)</td>
<td>P(b)</td>
</tr>
<tr>
<td>•</td>
<td>V_SXCVI: T(dNaN)</td>
<td>V_SXCVI: T(dNaN)</td>
<td>P(b)</td>
<td>V_SXSNAN: U(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>P(a)</td>
<td>V_SXSNAN: U(a)</td>
<td>V_SXSNAN: U(a)</td>
</tr>
<tr>
<td>SNaN</td>
<td>V_SXSNAN: U(a)</td>
<td>V_SXSNAN: U(a)</td>
<td>V_SXSNAN: U(a)</td>
<td>V_SXSNAN: U(a)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- See next table.
- dINF Default infinity
- dNaN Default quiet NaN
- Fn Finite nonzero numbers (includes both subnormal and normal numbers)
- P(x) The QNaN of operand x is propagated and placed in FRT[p]
- T(x) The value x is placed in FRT[p]
- U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- V_SXCVI The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions)
- V_SXSNAN The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions)

### Figure 100. Actions (part 2) Quantize

<table>
<thead>
<tr>
<th>Te &lt; Se</th>
<th>V_b &gt; (10^p - 1) × 10^{16}</th>
<th>0</th>
<th>Fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Te = Se</td>
<td>V_b ≤ (10^p - 1) × 10^{16}</td>
<td>E(0)</td>
<td>E(0)</td>
</tr>
<tr>
<td>Te &gt; Se</td>
<td>E(0)</td>
<td>E(0)</td>
<td>Q(b)</td>
</tr>
</tbody>
</table>

**Explanation:**
- dNaN Default quiet NaN
- E(0) The value of zero with the exponent value Te is placed in FRT[p].
- L(x) The operand x is converted to the form with the exponent value Te.
- p The precision of the format.
- QR(x) The operand x is rounded to the result of the form with the exponent value Te based on the specified rounding mode. The result of that form is placed in FRT[p].
- Se The exponent of the operand in FRA[p].
- Te The target exponent; FRA[p] for dqua[q], or TE, a 5-bit signed binary integer for dqua[q].
- T(x) The value x is placed in FRT[p].
- V_b The value of the operand in FRB[p].
- W(x) The value and the form of operand x is placed in FRT[p].
- V_SXCVI: The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions.)
DFP Reround [Quad]  Z23-form

<table>
<thead>
<tr>
<th>drrnd</th>
<th>FRT, FRA, FRB, RMC</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>drrnd.</td>
<td>FRT, FRA, FRB, RMC</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>FRB</th>
<th>RMC</th>
<th>35</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>FRA</th>
<th>FRBp</th>
<th>RMC</th>
<th>35</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

Let k be the contents of bits 58:63 of FRA that specifies the reference significance.

When the DFP operand in FRB[p] is a finite number, and if the reference significance is zero, or if the reference significance is nonzero and the number of significant digits of the source operand is less than or equal to the reference significance, then the value and the form of the source operand is placed in FRT[p]. If the reference significance is nonzero and the number of significant digits of the source operand is greater than the reference significance, then the source operand is converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field. The result of the form with the specified number of significant digits is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p].

For this instruction, the number of significant digits of the value 0 is considered to be zero. The ideal exponent is the greater value of the exponent of the operand in FRB[p] and the referenced exponent. The referenced exponent is the resultant exponent if the operand in FRB[p] would have been converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field.

If the exponent of the rounded result of the form that has the specified number of significant digits would be greater than $X_{\text{max}}$, an invalid operation exception (VXCVI) occurs. When the invalid-operation exception occurs, and if the exception is disabled, a default QNaN is returned. When an invalid-operation exception occurs, no inexact exception is recognized.

In the absence of an invalid-operation exception, if the result differs in value from the operand in FRB[p], an inexact exception is recognized.

This operation causes neither an overflow nor an underflow exception.

Figure 102 summarizes the actions for Reround. The table does not include the setting of the FPSR<sub>FPFR</sub> field. The FPSR<sub>FPFR</sub> field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:
- FPRF
- FR
- FI
- FX
- XX
- VXSNAN
- VXCVI
- CR1 (if Rc=1)

Programming Note

DFP Reround can be used to adjust a DFP value (FRB[p]) to have no more than a specified number (FRA[p]58:63) of significant digits. The result (FRT[p]) is right-justified leaving the specified number of digits and rounded as specified by the RMC field. If rounding increases the number of significant digits, the result is adjusted again (the significand is shifted right 1 digit and the exponent is incremented by 1). Figure 101 has example results from DFP Reround for 1, 2, and 10 significant digits.

Programming Note

DFP Reround is primarily used to round a DFP value to a specific number of digits before conversion to string format for printing or display. Another use for DFP Reround is to obtain the effective exponent of the most significant digit by specifying a reference significance of 1. The exponent can be extracted and used to compute the number of significant digits or to left-justify a value.

For example, the following sequence computes the number of significant digits and returns it as an integer. FRB is the DFP value for which we want the number of significant digits; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for doublewords at offsets -8 and -16. These doublewords are used to transfer the biased exponents from the FPRs to GPRs for integer computation. R3 contains the result of $E(\text{reround}(1,FRA)) - E(FRA) + 1$, where $E(x)$ represents the biased exponent of x.

```
dxex  f0,FRB
stfd  f0,-16(r1)
drrnd  f1,f13,FRB,1  # reround 1 digit toward 0
dxex  f1,f1
stfd  f1,-8(r1)
lfd  r11,-16(r1)
lfd  r3,-8(r1)
subf  r3,r11,r3
addi  r3,r3,1
```

Given the value 412.34 the result is $E(4 \times 102) - E(41234 \times 10-2) + 1 = (398+2) - (398-2) + 1 = 400 - 396 + 1 = 5$. Additional code is required to detect and handle special values like Subnormal, Infinity, and NaN.
### Programming Note

*DFP Reround* combined with *DFP Quantize* can be used to left justify a value (as needed by the frexp function). FRB is the DFP value for which we want to left justify; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for a doubleword at offset -8. This doubleword is used to transfer the biased exponents from the FPR to a GPR, for integer computation. The adjusted biased exponent (+ format precision - 1) is transferred back into an FPR so it can be inserted into the rerounded value. The adjusted rerounded value becomes the quantize reference value. The quantize instruction returns the left justified result in FRT.

```
drrmd f1, f13, FRB, 1 # reround 1 digit toward 0
dxex f0, f1
stfd f0, -8(r1)
ldr r1l, -8(r1)
addi r1l, r1l, 15 # biased exp + precision - 1
ldr r1l, -8(r1)
stfd f0, -8(r1)
diex f1, f0, f1 # adjust exponent
dqua FRT, f1, f0, 1 # quantize to adjusted exponent
```
### Actions for Reround when operand b in FRB[p] is

<table>
<thead>
<tr>
<th>Condition</th>
<th>0*</th>
<th>Fn</th>
<th>∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>k ≠ 0, k &lt; m</td>
<td>-</td>
<td>RR(b) or</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_{XCVI}; T(dNaN)</td>
<td></td>
<td></td>
<td>V_{XSNAN}; U(b)</td>
</tr>
<tr>
<td>k ≠ 0, k = m</td>
<td>-</td>
<td>W(b)</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td></td>
</tr>
<tr>
<td>k ≠ 0 and k &gt; m, or k = 0</td>
<td>W(b)</td>
<td>W(b)</td>
<td>T(dINF)</td>
<td>P(b)</td>
<td>V_{XSNAN}; U(b)</td>
</tr>
</tbody>
</table>

**Explanation:**
- * The number of significant digits of the value 0 is considered to be zero for this instruction.
- - Not applicable.
- dINF Default infinity.
- Fn Finite nonzero numbers (includes both subnormal and normal numbers).
- k Reference significance, which specifies the number of significant digits in the target operand.
- m Number of significant digits in the operand in FRB[p].
- P(x) The QNaN of operand x is propagated and placed in FRT[p].
- RR(x) The value x is rounded to the form that has the specified number of significant digits.
  - If RR(x) \( \leq (10^{k-1}) \times 10^{\lambda_{\text{max}}} \), then RR(x) is returned; otherwise an invalid-operation exception is recognized.
- T(x) The value x is placed in FRT[p].
- U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
- V_{XCVI} The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions.)
- V_{XSNAN} The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. See Section 6.5.10.1 for actions.
- W(x) The value and the form of x is placed in FRT[p].

**Figure 102.** Actions: Reround
**DFP Round To FP Integer With Inexact [Quad]**

The DFP operand in FRB[p] is rounded to a floating-point integer and placed into FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the larger value of zero and the exponent of the operand in FRB[p].

The rounding mode used is specified in the RMC field. When the RMC-encoding-selection (R) bit is zero, the RMC field contains the primary encoding; when the bit is one, the field contains the secondary encoding.

In addition to coercion of the converted value to fit the target format, the special rounding used by Round To FP Integer also coerces the target exponent to the ideal exponent.

When the operand in FRB[p] is a finite number and the exponent is less than zero, the operand is rounded to the result with an exponent of zero. When the exponent is greater than or equal to zero, the result is set to the numerical value and the form of the operand in FRB[p].

When the result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 103 summarizes the actions for Round To FP Integer With Inexact. The table does not include the setting of the FPSCR_FPRF field. The FPSCR_FPRF field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

**Special Registers Altered:**
- FPRF
- FR
- FI
- FX
- VXSNAN
- CR1 (if R=1)

---

**Programming Note**

The DFP Round To FP Integer With Inexact and DFP Round To FP Integer With Inexact Quad instructions can be used to implement the decimal equivalent of the C99 rint function by specifying the primary RMC encoding for round according to FPSCR_DIR (R=0, RMC=11). The specification for rint requires the inexact exception be raised if detected.
<table>
<thead>
<tr>
<th>Operand b in FRB is</th>
<th>Is n not precise ((n \neq b))</th>
<th>Inv.-Op. Exception Enabled</th>
<th>Inexact Exception Enabled</th>
<th>Is n Incremented ((lnl &gt; lbl))</th>
<th>Actions*</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>No(^1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(T(-dINF), FI \leftarrow 0, FR \leftarrow 0)</td>
</tr>
<tr>
<td>(F)</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(W(n), FI \leftarrow 0, FR \leftarrow 0)</td>
</tr>
<tr>
<td>(F)</td>
<td>Yes</td>
<td>-</td>
<td>No</td>
<td>No</td>
<td>(W(n), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1)</td>
</tr>
<tr>
<td>(F)</td>
<td>Yes</td>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>(W(n), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1)</td>
</tr>
<tr>
<td>(F)</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>No</td>
<td>(W(n), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TX)</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>No(^1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(T(+dINF), FI \leftarrow 0, FR \leftarrow 0)</td>
</tr>
<tr>
<td>QNaN</td>
<td>No(^1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(P(b), FI \leftarrow 0, FR \leftarrow 0)</td>
</tr>
<tr>
<td>SNaN</td>
<td>No(^1)</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>(U(b), FI \leftarrow 0, FR \leftarrow 0, VXSNAN \leftarrow 1)</td>
</tr>
<tr>
<td>SNaN</td>
<td>No(^1)</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>VXSNAN \leftarrow 1, TV</td>
</tr>
</tbody>
</table>

Explanation:

* Setting of XX and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR is part of the exception actions. (See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)

- The actions do not depend on this condition.

1 This condition is true by virtue of the state of some condition to the left of this column.

dINF Default infinity.

F All finite numbers, including zeros.

FI Floating-Point-Fraction-Inexact status flag, FPSCR\(_{FI}\).

FR Floating-Point-Fraction-Rounded status flag, FPSCR\(_{FR}\).

n The value derived when the source operand, b, is rounded to an integer using the special rounding for Round To FP Integer.

P(x) The QNaN of operand x is propagated and placed in FRT\([p]\).

T(x) The value x is placed in FRT\([p]\).

TV The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.

TX The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.

U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FPT\([p]\).

W(x) The value x in the form of zero exponent or the source exponent is placed in FRT\([p]\).

XX Floating-Point-Inexact-Exception status flag, FPSCR\(_{XX}\).

Figure 103. Actions: Round to FP Integer With Inexact
**DFP Round To FP Integer Without Inexact**

**Z23-form**

<table>
<thead>
<tr>
<th>drintn</th>
<th>R,FRT,FRB,RMC</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>drintn.</td>
<td>R,FRT,FRB,RMC</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>FRT /// R FRB RMC</th>
<th>227</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 11 16</td>
<td>23</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>drintnq</th>
<th>R,FRTp,FRBp,RMC</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>drintnq</td>
<td>R,FRTp,FRBp,RMC</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp /// R FRBp RMC</th>
<th>227</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 11 15 21 23</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

This operation is the same as the *Round To FP Integer With Inexact* operation, except that this operation does not recognize an inexact exception.

Figure 104 summarizes the actions for *Round To FP Integer Without Inexact*. The table does not include the setting of the FPSCR[RPF] field. The FPSCR[RPF] field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

Special Registers Altered:
- FPRF FR (set to 0) FI (set to 0)
- FX
- VXSNAN
- CR1

**Programming Note**
The *DFP Round To FP Integer Without Inexact* and *DFP Round To FP Integer Without Inexact Quad* instructions can be used to implement decimal equivalents of several C99 rounding functions by specifying the appropriate R and RMC field values.

### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>R</th>
<th>RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ceil</td>
<td>1</td>
<td>0b00</td>
</tr>
<tr>
<td>Floor</td>
<td>1</td>
<td>0b01</td>
</tr>
<tr>
<td>Nearbyint</td>
<td>0</td>
<td>0b11</td>
</tr>
<tr>
<td>Round</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>Trunc</td>
<td>0</td>
<td>0b01</td>
</tr>
</tbody>
</table>

Note that nearbyint is similar to the rint function but without raising the inexact exception. Similarly ceil, floor, round, and trunc do not require the inexact exception.

### Operand b in FRB is

<table>
<thead>
<tr>
<th>Inv.-Op. Exception Enabled</th>
<th>Actions*</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>T(-dINF), FI ← 0, FR ← 0</td>
</tr>
<tr>
<td>F</td>
<td>W(n), FI ← 0, FR ← 0</td>
</tr>
<tr>
<td>+∞</td>
<td>T(+dINF), FI ← 0, FR ← 0</td>
</tr>
<tr>
<td>QNaN</td>
<td>P(b), FI ← 0, FR ← 0</td>
</tr>
<tr>
<td>SNaN</td>
<td>U(b), FI ← 0, FR ← 0, VXSNAN=1</td>
</tr>
<tr>
<td>SNaN</td>
<td>VXSNAN ← 1, TV</td>
</tr>
</tbody>
</table>

Explanation:
- Setting of VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Invalid Operation Exception” for more details.)
- The actions do not depend on this condition.
- dINF Default infinity.
- F All finite numbers, including zeros.
- FI Floating-Point-Fraction-Inexact status flag, FPSCR[FI].
- FR Floating-Point-Fraction-Rounded status flag, FPSCR[FR].
- n The value derived when the source operand, b, is rounded to an integer using the special rounding for Round-To-FP-Integer.
- P(x) The QNaN of operand x is propagated and placed in FRT[p].
- T(x) The value x is placed in FRT[p].
- TV The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FEO and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
- U(x) The SNaN of operand x is converted to the corresponding QNaN and placed in FPT[p].
- W(x) The value x in the form of zero exponent or the source exponent is placed in FRT[p].

**Figure 104. Actions: Round to FP Integer Without Inexact**
6.6.5 DFP Conversion Instructions

The DFP conversion instructions consist of data-format conversion instructions and data-type conversion instructions. They are all X-form instructions and employ the record bit (Rc).

6.6.5.1 DFP Data-Format Conversion Instructions

The data-format conversion instructions consist of Convert To DFP Long, Convert To DFP Extended, Round To DFP Short, and Round To DFP Long. Figure 105 summarizes the actions for these instructions.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

--- Programming Note ---

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP Load Floating-Point Single and Store Floating-Point Single functions, the widening is performed by loading the DFP short value with a Load Floating as Integer Word Indexed followed by a DFP Convert to DFP Long, and narrowing is performed by a DFP Round to DFP Short followed by a Store Floating-Point as Integer Word Indexed. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.
### DFP Convert To DFP Long  

<table>
<thead>
<tr>
<th>X-form</th>
<th>DCTDP</th>
<th>FRT,FRB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dctdp</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td></td>
<td>dctdp.</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The DFP short operand in bits 32:63 of FRB is converted to DFP long format and the converted result is placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP long format and does not cause an invalid-operation exception.

**Special Registers Altered:**
- FPRF FR (undefined)
- FI (undefined)
- CR1 (if Rc=1)

**Programming Note**

Note that DFP short format is a storage-only format. Therefore, conversion of a short SNaN to long format will not cause an exception and the SNaN is preserved. Subsequent operation on that SNaN in long format will cause an exception.

### DFP Convert To DFP Extended  

<table>
<thead>
<tr>
<th>X-form</th>
<th>DCTQPQ</th>
<th>FRTp,FRB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dctqpq</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td></td>
<td>dctqpq.</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

The DFP long operand in the FRB is converted to DFP extended format and placed into FRTp. The sign of the result is the same as the sign of the operand in FRB. The ideal exponent is the exponent of the operand in FRB.

If the operand in FRB is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP extended format.

**Special Registers Altered:**
- FPRF FR (set to 0)
- FI (set to 0)
- CR1 (if Rc=1)
- FX
- VXSNAN
**DFP Round To DFP Short**  
**X-form**

<table>
<thead>
<tr>
<th>drsp</th>
<th>FRT,FRB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>drsp.</td>
<td>FRT,FRB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRB</th>
<th>770</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

The DFP long operand in FRB is converted and rounded to DFP short format. The DFP short value is extended on the left with zeros to form a 64-bit entity and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP short format and does not cause an invalid-operation exception.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

**Special Registers Altered:**
- FPRF
- FR
- FI
- FX
- OX
- UX
- XX
- CR1 (if Rc=1)

**Programming Note**

Note that DFP short format is a storage-only format. Therefore, conversion of a long SNaN to short format will not cause an exception. Converting a long format SNaN to short format is an implied move operation.

---

**DFP Round To DFP Long**  
**X-form**

<table>
<thead>
<tr>
<th>drdpq</th>
<th>FRTp,FRBp</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>drdpq.</td>
<td>FRTp,FRBp</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>FRBp</th>
<th>770</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

The DFP extended operand in FRBp is converted and rounded to DFP long format. The result concatenated with 64 0s is placed in FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the operand in FRBp.

If the operand in FRBp is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP long format.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

**Special Registers Altered:**
- FPRF
- FR
- FI
- FX
- OX
- UX
- XX
- VXSNAN
- CR1 (if Rc=1)

**Programming Note**

Note that DFP Round to DFP Long, while producing a result in DFP long format, actually targets a register pair, writing 64 0s in FRTp+1.
6.6.5.2 DFP Data-Type Conversion Instructions

The DFP data-type conversion instructions are used to convert data type between DFP and fixed.

### DFP Convert From Fixed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dcffix</code></td>
<td><code>FRT,FRB</code></td>
<td>(Rc=0)</td>
</tr>
<tr>
<td><code>dcffix.</code></td>
<td><code>FRT,FRB</code></td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>//</th>
<th>FRB</th>
<th>802</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Long value and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

The FPSCR_FPRF field is set to the class and sign of the result.

**Special Registers Altered:**
- FPRF FR FI
- FX XX
- CR1 (if Rc=1)

### DFP Convert From Fixed Quad

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dcffixq</code></td>
<td><code>FRTp,FRB</code></td>
<td>(Rc=0)</td>
</tr>
<tr>
<td><code>dcffixq.</code></td>
<td><code>FRTp,FRB</code></td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>63</th>
<th>FRTp</th>
<th>//</th>
<th>FRB</th>
<th>802</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

The FPSCR_FPRF field is set to the class and sign of the result.

**Special Registers Altered:**
- FPRF (undefined) FR FI
- FX XX
- VXSNAN VXCVI
- CR1 (if Rc=1)

### DFP Convert To Fixed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dctfix</code></td>
<td><code>FRT,FRB</code></td>
<td>(Rc=0)</td>
</tr>
<tr>
<td><code>dctfix.</code></td>
<td><code>FRT,FRB</code></td>
<td>(Rc=1)</td>
</tr>
<tr>
<td><code>dctfixq</code></td>
<td><code>FRTp,FRBp</code></td>
<td>(Rc=0)</td>
</tr>
<tr>
<td><code>dctfixq.</code></td>
<td><code>FRTp,FRBp</code></td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>//</th>
<th>FRB</th>
<th>290</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

The DFP operand in FRB[p] is rounded to an integer value and is placed into FRT in the 64-bit signed binary integer format. The sign of the result is the same as the sign of the source operand, except when the source operand is a NaN or a zero.

The FPSCR_FPRF field is set to the class and sign of the result.

**Special Registers Altered:**
- FPRF (undefined) FR FI
- VXSNAN VXCVI
- CR1 (if Rc=1)

### Programming Note

It is recommended that software pre-round the operand to a floating-point integral using `drintx[q]` or `drintn[q]` is a rounding mode other than the current rounding mode specified by FPSCR_DRN is needed. Saving, modifying and restoring the FPSCR just to temporarily change the rounding mode is less efficient than just employing drint[p] or drint[p] which override the current rounding mode using an immediate control field.

For example if the desired function rounding is Round to Nearest, Ties away from 0 but the default rounding (from FPSCR_DRN) is Round to Nearest, Ties to Even then following is preferred.

- `drintn 0, f1, f1, 2`
- `dctfix f1, f1`
| Operand $b$ in FRB[p] is | $q$ is | Is $n$ not precise \((n \neq b)\) | Inv.-Op. Except. Enabled | Inexact Except. Enabled | Is $n$ Incremented \(|n| > |b|\) | Actions * |
|--------------------------|--------|-------------------------------|-------------------|-------------------|-------------------------|-------------|
| $-\infty \leq b < MN$    | $< MN$ | -                             | No                | -                 | -                       | $T(MN), FI \leftarrow 0, FR \leftarrow 0, VXCVI \leftarrow 1$ |
| $-\infty \leq b < MN$    | $< MN$ | -                             | Yes               | -                 | -                       | $VXCVI \leftarrow 1, TV$ |
| $-\infty < b < MN$       | $= MN$ | -                             | No                | -                 | -                       | $T(MN), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1$ |
| $-\infty < b < MN$       | $= MN$ | -                             | -                 | Yes               | -                       | $T(MN), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1, TX$ |
| $MN \leq b < 0$          | -      | No                            | -                 | -                 | -                       | $T(n), FI \leftarrow 0, FR \leftarrow 0$ |
| $MN \leq b < 0$          | -      | Yes                           | No                | No                | No                      | $T(n), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1$ |
| $MN \leq b < 0$          | -      | Yes                           | -                 | Yes               | No                      | $T(n), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1$ |
| $MN \leq b < 0$          | -      | Yes                           | -                 | Yes               | Yes                     | $T(n), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1, TX$ |
| $\pm 0$                  | -      | No                            | -                 | -                 | -                       | $T(0), FI \leftarrow 0, FR \leftarrow 0$ |
| $0 < b \leq MP$          | -      | No                            | -                 | -                 | -                       | $T(n), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1$ |
| $0 < b \leq MP$          | -      | Yes                           | No                | No                | No                      | $T(n), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1$ |
| $0 < b \leq MP$          | -      | Yes                           | -                 | Yes               | Yes                     | $T(n), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1, TX$ |
| $0 < b \leq MP$          | -      | Yes                           | -                 | Yes               | Yes                     | $T(n), FI \leftarrow 1, FR \leftarrow 1, XX \leftarrow 1, TX$ |
| $MP < b < +\infty$       | $= MP$ | -                             | No                | -                 | -                       | $T(MP), FI \leftarrow 1, FR \leftarrow 0, XX \leftarrow 1$ |
| $MP < b < +\infty$       | $= MP$ | -                             | Yes               | -                 | -                       | $T(MP), FI \leftarrow 1, FR \leftarro 0, XX \leftarrow 1, TX$ |
| $MP < b < +\infty$       | $> MP$ | -                             | No                | -                 | -                       | $MP \leftarrow 0, FR \leftarrow 0, VXCVI \leftarrow 1$ |
| QNaN                     | -      | No                            | -                 | -                 | -                       | $T(MN), FI \leftarrow 0, FR \leftarrow 0, VXCVI \leftarrow 1$ |
| QNaN                     | -      | Yes                           | -                 | -                 | -                       | $VXCVI \leftarrow 1, TX$ |
| SNaN                     | -      | No                            | -                 | -                 | -                       | $T(MN), FI \leftarrow 0, FR \leftarrow 0, VXCVI \leftarrow 1, VXSNAN \leftarrow 1$ |
| SNaN                     | -      | Yes                           | -                 | -                 | -                       | $VXCVI \leftarrow 1, VXSNAN \leftarrow 1, TX$ |

Explanation:

* Setting of XX, VXCVI, and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)

- The actions do not depend on this condition.

| FI | Floating-Point-Fraction-Inexact status flag, FPSCRFI. |
| FR | Floating-Point-Fraction-Rounded status flag, FPSCRFR. |
| MN | Maximum negative number representable by the 64-bit binary integer format |
| MP | Maximum positive number representable by the 64-bit binary integer format |
| n  | The value $q$ converted to a fixed-point result. |
| q  | The value derived when the source value $b$ is rounded to an integer using the specified rounding mode |
| T(x) | The value $x$ is placed in FRT[p]. |
| TV | The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode. |
| TX | The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode. |
| VXCVI | The FPSCR$_{VXCVI}$ invalid operation exception status bit. |
| VXSNAN | The FPSCR$_{VXSNAN}$ invalid operation exception status bit. |
| XX | Floating-Point-Inexact-Exception status flag, FPSCR$_{XX}$. |

Figure 106. Actions: Convert To Fixed
6.6.6 DFP Format Instructions

The DFP format instructions are used to compose or decompose a DFP operand. A source operand of SNaN does not cause an invalid-operation exception. All format instructions employ the record bit (Rc).

**DFP Decode DPD To BCD [Quad] X-form**

<table>
<thead>
<tr>
<th>ddedpd</th>
<th>SP,FRT,FRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddedpd.</td>
<td>SP,FRT,FRB</td>
</tr>
<tr>
<td>59</td>
<td>FRT</td>
</tr>
<tr>
<td>6</td>
<td>SP</td>
</tr>
<tr>
<td>11</td>
<td>///</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>322</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

**DFP Encode BCD To DPD [Quad] X-form**

<table>
<thead>
<tr>
<th>denbcd</th>
<th>S,FRT,FRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>denbcd.</td>
<td>S,FRT,FRB</td>
</tr>
<tr>
<td>59</td>
<td>FRT</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>///</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>834</td>
</tr>
<tr>
<td>31</td>
<td>Rc</td>
</tr>
</tbody>
</table>

A portion of the significand of the DFP operand in FRB[p] is converted to a signed or unsigned BCD number depending on the SP field. For infinity and NaN, the significand is considered to be the contents in the trailing significand field padded on the left by a zero digit.

**SP0 = 0 (unsigned conversion)**

The rightmost 16 digits of the significand (32 digits for ddedpdq) is converted to an unsigned BCD number and the result is placed into FRT[p].

**SP0 = 1 (signed conversion)**

The rightmost 15 digits of the significand (31 digits for ddedpdq) is converted to a signed BCD number with the same sign as the DFP operand, and the result is placed into FRT[p]. If the DFP operand is negative, the sign is encoded as 0b1101. If the DFP operand is positive, SP1 indicates which preferred plus sign encoding is used. If SP1 = 0, the plus sign is encoded as 0b1100 (the option-1 preferred sign code), otherwise the plus sign is encoded as 0b1111 (the option-2 preferred sign code).

**Special Registers Altered:**

CR1 (if Rc=1)

The format instructions consist of Decode DPD To BCD, Encode BCD To DPD, Extract Biased Exponent, Insert Biased Exponent, Shift Significand Left Immediate, and Shift Significand Right Immediate.
### DFP Extract Biased Exponent [Quad] X-form

| dxex  | FRT,FRB | (Rc=0) |
| dxex. | FRT,FRB | (Rc=1) |

| dxexq | FRT,FRBp | (Rc=0) |
| dxexq. | FRT,FRBp | (Rc=1) |

The biased exponent of the operand in FRB[p] is extracted and placed into FRT in the 64-bit signed binary integer format. When the operand in FRB is an infinity, QNaN, or SNaN, a special code is returned.

**Operand Result**
- Finite Number: biased exponent value
- Infinity: -1
- QNaN: -2
- SNaN: -3

**Special Registers Altered:**
- CR1 (if Rc=1)

**Programming Note**
The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

### DFP Insert Biased Exponent [Quad] X-form

| diex  | FRT,FRA,FRB | (Rc=0) |
| diex. | FRT,FRA,FRB | (Rc=1) |

| diexq | FRTp,FRA,FRBp | (Rc=0) |
| diexq. | FRTp,FRA,FRBp | (Rc=1) |

Let a be the value of the 64-bit signed binary integer in FRA.

- **Result**
  - \( a > \text{MBE} \): QNaN
  - \( MBE \geq a \geq 0 \): Finite number with biased exponent a
  - \( a = -1 \): Infinity
  - \( a = -2 \): QNaN
  - \( a = -3 \): SNaN
  - \( a < -3 \): QNaN

1 Maximum biased exponent for the target format

When \( 0 \leq a \leq \text{MBE} \), a is the biased target exponent that is combined with the sign bit and the significand value of the DFP operand in FRB[p] to form the DFP result in FRT[p]. The ideal exponent is the specified target exponent.

When a specifies a special code (\( a < 0 \) or \( a > \text{MBE} \)), an infinity, QNaN, or SNaN is formed in FRT[p] with the trailing significand field containing the value from the trailing significand field of the source operand in FRB[p], and with an N-bit combination field set as follows:

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.

- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-5 bits are set to zero.

- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-5 bits are set to zero.

**Special Registers Altered:**
- CR1 (if Rc=1)

**Programming Note**
The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.
### Table: Actions for Insert Biased Exponent

<table>
<thead>
<tr>
<th>Operand a in FRA[p] specifies</th>
<th>Actions for Insert Biased Exponent when operand b in FRB[p] specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>N, Rb</td>
</tr>
<tr>
<td>( \infty )</td>
<td>I, Rb</td>
</tr>
<tr>
<td>QNaN</td>
<td>Q, Rb</td>
</tr>
<tr>
<td>SNaN</td>
<td>S, Rb</td>
</tr>
</tbody>
</table>

**Explanation:**

- **F**: All finite numbers, including zeros
- **I**: The combination field in FRT[p] is set to indicate a default Infinity.
- **N**: The combination field in FRT[p] is set to the specified biased exponent in FRA and the leftmost significand digit in FRB[p].
- **Q**: The combination field in FRT[p] is set to indicate a default QNaN.
- **S**: The combination field in FRT[p] is set to indicate a default SNaN.
- **Z**: The combination field in FRT[p] is set to indicate the specific biased exponent in FRA and a leftmost coefficient digit of zero.
- **Rb**: The contents of the trailing significand field in FRB[p] are reencoded using preferred DPD encodings and the reencoded result is placed in the same field in FRT[p]. The sign bit of FRB[p] is copied into the sign bit in FRT[p].

**Figure 107. Actions: Insert Biased Exponent**
DFP Shift Significand Left Immediate
[Quad] Z22-form

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>SH</th>
<th>66</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>22</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

dscli  FRT,FRA,SH  (Rc=0)
dscli. FRT,FRA,SH  (Rc=1)
dscliq FRTp,FRAp,SH  (Rc=0)
dscliq. FRTp,FRAp,SH  (Rc=1)

The significand of the DFP operand in FRA[p] is shifted left SH digits. For a NaN or infinity, all significand digits are in the trailing significand field. SH is a 6-bit unsigned binary integer. Digits shifted out of the leftmost digit are lost. Zeros are supplied to the vacated positions on the right. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format's N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

Special Registers Altered:
CR1  (if Rc=1)

DFP Shift Significand Right Immediate
[Quad] Z22-form

<table>
<thead>
<tr>
<th>59</th>
<th>FRT</th>
<th>FRA</th>
<th>SH</th>
<th>98</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>22</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

dscri  FRT,FRA,SH  (Rc=0)
dscri. FRT,FRA,SH  (Rc=1)
dscriq FRTp,FRAp,SH  (Rc=0)
dscriq. FRTp,FRAp,SH  (Rc=1)

The significand of the DFP operand in FRA[p] is shifted right SH digits. For a NaN or infinity, all significand digits are in the trailing significand field. SH is a 6-bit unsigned binary integer. Digits shifted out of the units digit are lost. Zeros are supplied to the vacated positions on the left. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format's N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

Special Registers Altered:
CR1  (if Rc=1)
### 6.6.7 DFP Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Full Name</th>
<th>FORM</th>
<th>Operands</th>
<th>SNaNNs</th>
<th>Encoding</th>
<th>FPRF</th>
<th>FPCC</th>
<th>Frp</th>
<th>Opx</th>
<th>Upx</th>
<th>Xpx</th>
<th>Ext Rm</th>
<th>IE</th>
<th>Ce</th>
</tr>
</thead>
<tbody>
<tr>
<td>dadd</td>
<td>DFP Add</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>daddq</td>
<td>DFP Add Quad</td>
<td>X</td>
<td>FRTp, Fra, FRBp</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsup</td>
<td>DFP Subtract</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsupq</td>
<td>DFP Subtract Quad</td>
<td>X</td>
<td>FRTp, Fra, FRBp</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmul</td>
<td>DFP Multiply</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmulq</td>
<td>DFP Multiply Quad</td>
<td>X</td>
<td>FRTp, Fra, FRBp</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddv</td>
<td>DFP Divide</td>
<td>X</td>
<td>FRT, FRA, FRB</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ddvq</td>
<td>DFP Divide Quad</td>
<td>X</td>
<td>FRTp, Fra, FRBp</td>
<td>Y</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>V</td>
<td>O</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### dcmpo
- DFP Compare Ordered: X | BF, FRA, FRB | Y - - N Y V - - N

#### dcmpoq
- DFP Compare Ordered Quad: X | BF, FRA, FRBp | Y - - N Y V - - N

#### dcmpoq
- DFP Compare Ordered: X | BF, FRA, FRB | Y - - N Y V - - N

#### dcmpoq
- DFP Compare Ordered Quad: X | BF, FRA, FRBp | Y - - N Y V - - N

#### dtsdc
- DFP Test Data Class: Z22 | BF, FRA, DCM | N - - N Y1 - - N

#### dtsdcq
- DFP Test Data Class Quad: Z22 | BF, Fra, DCM | N - - N Y1 - - N

#### dtsdg
- DFP Test Data Group: Z22 | BF, FRA, DCM | N - - N Y1 - - N

#### dtsdgq
- DFP Test Data Group Quad: Z22 | BF, Fra, DCM | N - - N Y1 - - N

#### dtstx
- DFP Test Exponent: X | BF, FRA, FRB | Y - - N Y V - - N

#### dtstxq
- DFP Test Exponent Quad: X | BF, Fra, FRBp | Y - - N Y V - - N

#### dtsf
- DFP Test Significance: X | BF, Fra(Fix), FRB | Y - - N Y V - - N

#### dtsfq
- DFP Test Significance Quad: X | BF, Fra(Fix), FRBp | Y - - N Y V - - N

#### dqua
- DFP Quantize: Z23 | TE, FRT, FRB, RMC | Y N RE Y V Y Y Y

#### dquaq
- DFP Quantize Quad: Z23 | TE, FRTp, FRBp, RMC | Y N RE Y V Y Y Y

#### dquaf
- DFP Quantize: Z23 | BF, Fra, FRB | Y - - N Y V - - N

#### dquaqf
- DFP Quantize Quad: Z23 | BF, Fra, FRBp | Y - - N Y V - - N

#### dquas
- DFP Quantize: Z23 | BF, Fra, FRB | Y - - N Y V - - N

#### dquasq
- DFP Quantize Quad: Z23 | BF, Fra, FRBp | Y - - N Y V - - N

#### dquasf
- DFP Quantize: Z23 | BF, Fra(Fix), FRB | Y - - N Y V - - N

#### dquasfq
- DFP Quantize Quad: Z23 | BF, Fra(Fix), FRBp | Y - - N Y V - - N

#### dquaf
- DFP Quantize: Z23 | BF, Fra, FRB | Y - - N Y V - - N

#### dquafq
- DFP Quantize Quad: Z23 | BF, Fra, FRBp | Y - - N Y V - - N

#### dquasf
- DFP Quantize: Z23 | BF, Fra(Fix), FRB | Y - - N Y V - - N

#### dquasfq
- DFP Quantize Quad: Z23 | BF, Fra(Fix), FRBp | Y - - N Y V - - N

#### dceq
- DFP Convert To DFP Long: X | FRT, FRB (DFP Short) | Y N RE Y V Y Y Y

#### dceq
- DFP Convert To DFP Extended: X | FRTp, FRB | Y N RE Y V Y Y Y

#### dsr
- DFP Round To DFP Short: X | FRT (DFP Short), FRB | Y N RE Y V Y Y Y

#### dsrq
- DFP Round To DFP Long: X | FRTp, FRBp | Y N RE Y V O U X Y Y Y

#### dcffixq
- DFP Convert From Fixed Quad | X | FRTp, FRB (Fix) | - N RE Y Y U Y Y

#### dcffixq
- DFP Convert To Fixed: X | FRT(FIX), FRB | Y N - U U V X Y - Y

#### dcffixq
- DFP Convert To Fixed Quad | X | FRT (FIX), FRBp | Y N - U U V X Y - Y

#### ddedf
- DFP Decode DPD To BCD: X | SP, FRT(BCD), FRB | N - - N N - - Y

---

**Figure 108: Decimal Floating-Point Instructions Summary**
### Table: Decimal Floating-Point Instructions Summary (Continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Full Name</th>
<th>FORM</th>
<th>Operands</th>
<th>SNaN</th>
<th>VS</th>
<th>G</th>
<th>Encoding</th>
<th>FPSCR</th>
<th>Exception</th>
<th>IE</th>
<th>RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddedpdq</td>
<td>DFP Decode DPD To BCD Quad</td>
<td>X</td>
<td>SP, FRTp(BCD), FRBp</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>denbcd</td>
<td>DFP Encode BCD To DPD</td>
<td>X</td>
<td>S, FRT, FRB (BCD)</td>
<td>-</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>Y</td>
<td>V</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>denbcdq</td>
<td>DFP Encode BCD To DPD Quad</td>
<td>X</td>
<td>S, FRTp, FRBp (BCD)</td>
<td>-</td>
<td>N</td>
<td>RE</td>
<td>Y</td>
<td>Y</td>
<td>V</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>dxex</td>
<td>DFP Extract Biased Exponent</td>
<td>X</td>
<td>FRT (FIX), FRB</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dxexq</td>
<td>DFP Extract Biased Exponent Quad</td>
<td>X</td>
<td>FRT (FIX), FRBp</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>diex</td>
<td>DFP Insert Biased Exponent</td>
<td>X</td>
<td>FRT, FRA(FIX), FRB</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>diexq</td>
<td>DFP Insert Biased Exponent Quad</td>
<td>X</td>
<td>FRTp, FRA(FIX), FRBp</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>dscli</td>
<td>DFP Shift Significand Left Immediate</td>
<td>Z22</td>
<td>FRT,FRA,SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dscliq</td>
<td>DFP Shift Significand Left Immediate Quad</td>
<td>Z22</td>
<td>FRTp,FRAp,SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dscrri</td>
<td>DFP Shift Significand Right Immediate</td>
<td>Z22</td>
<td>FRT,FRA,SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>dscrriq</td>
<td>DFP Shift Significand Right Immediate Quad</td>
<td>Z22</td>
<td>FRTp,FRAp,SH</td>
<td>N</td>
<td>Y</td>
<td>RE</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

### Explanation:

1. FI and FR are set to zeros for these instructions.
2. A unique definition of the FPSCR<sub>FPCC</sub> field is provided for the instruction.
3. These are the only instructions that may generate an SNaN and also set the FPSCR<sub>FPRF</sub> field. Since the BFP FPSCR<sub>FPRF</sub> field does not include a code for SNaN, these instructions cause the need for redefining the FPSCR<sub>FPRF</sub> field for DFP.

- **DCM**: A 6-bit immediate operand specifying the data-class mask.
- **DGM**: A 6-bit immediate operand specifying the data-group mask.
- **G**: An SNaN can be generated as the target operand.
- **IE**: An ideal exponent is defined for the instruction.
- **FI**: Setting of the FPSCR<sub>F</sub> flag.
- **FR**: Setting of the FPSCR<sub>FR</sub> flag.
- **N**: No.
- **O**: An overflow exception may be recognized.
- **Rc**: The record bit, Rc, is provided to record FPSCR<sub>0:3</sub> in CR field 1.
- **RE**: The trailing significand field is reencoded using preferred DPD encodings. The preferred DPD encoding are also used for propagated NaNs, or converted NaNs and infinities.
- **RMC**: A 2-bit immediate operand specifying the rounding-mode control.
- **S**: An one-bit immediate operand specifying if the operation is signed or unsigned.
- **SP**: A two-bit immediate operand: one bit specifies if the operation is signed or unsigned and, for signed operations, another bit specifies which preferred plus sign code is generated.
- **U**: An underflow exception may be recognized.
- **V**: An invalid-operation exception may be recognized.
- **Vs**: An input operand of SNaN causes an invalid-operation exception.
- **X**: An inexact exception may be recognized.
- **Y**: Yes.
- **U**: Undefined
- **Z**: A zero-divide exception may be recognized.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

7.1 Introduction

7.1.1 Overview of the Vector-Scalar Extension

Category Vector-Scalar Extension (VSX) provides facilities supporting vector and scalar binary floating-point operations. The following VSX features are provided to increase opportunities for vectorization.

- A unified register file, a set of Vector-Scalar Registers (VSR), supporting both scalar and vector operations is provided, eliminating the overhead of vector-scalar data transfer through storage.
- Support for word-aligned storage accesses for both scalar and vector operations is provided.
- Robust support for IEEE-754 for both vector and scalar floating-point operations is provided.

Combining the Floating-Point Registers (FPR) defined in Chapter 4. Floating-Point Facility [Category: Floating-Point] and the Vector Registers (VR) defined in Chapter 5. Vector Facility [Category: Vector] provides additional registers to support more aggressive compiler optimizations for both vector and scalar operations.

Implementations of VSX must also implement the Floating-Point (Chapter 4) and Vector (Chapter 5) categories.

7.1.1.1 Compatibility with Category Floating-Point and Category Decimal Floating-Point Operations

The instruction sets defined in Chapter 4. Floating-Point Facility [Category: Floating-Point] and Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point] retain their definition with one primary difference. The FPRs are mapped to doubleword element 0 of VSRs 0-31. The contents of doubleword 1 of the VSR corresponding to a source FPR specified by an instruction are ignored. The contents of doubleword 1 of a VSR corresponding to the target FPR specified by an instruction are undefined.

Programming Note

Application binary interfaces extended to support VSX require special care of vector data written to VSRs 0-31 (i.e., VSRs corresponding to FPRs). Legacy scalar function calls employ doubleword-based loads and stores to preserve the contents of any nonvolatile registers. This has the adverse effect of not preserving the contents of doubleword 1 of these VSRs.

7.1.1.2 Compatibility with Category Vector Operations

The instruction set defined in Chapter 5. Vector Facility [Category: Vector], retains its definition with one primary difference. The VRs are mapped to VSRs 32-63.
7.2 VSX Registers

7.2.1 Vector-Scalar Registers

Sixty-four 128-bit VSRs are provided. See Figure 109. All VSX floating-point computations and other data manipulation are performed on data residing in Vector-Scalar Registers, and results are placed into a VSR.

Depending on the instruction, the contents of a VSR are interpreted as a sequence of equal-length elements (words or doublewords) or as a quadword. Each of the elements is aligned at its natural boundary within the VSR, as shown in Figure 109. Many instructions perform a given operation in parallel on all elements in a VSR. Depending on the instruction, a word element can be interpreted as a signed integer word (SW), an unsigned integer word (UW), a logical mask value (MW), or a single-precision floating-point value (SP); a doubleword element can be interpreted as a doubleword signed integer (SD), a doubleword unsigned integer (UD), a doubleword mask (DM), or a double-precision floating-point value (DP). In the instructions descriptions, phrases like *signed integer word element* are used as shorthand for *word element, interpreted as a signed integer*.

Load and Store instructions are provided that transfer a byte, halfword, word, doubleword, or quadword between storage and a VSR.

![Figure 109. Vector-Scalar Registers](image)

<table>
<thead>
<tr>
<th>VSR[0]</th>
<th>VSR[1]</th>
<th>...</th>
<th>VSR[62]</th>
<th>VSR[63]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 109. Vector-Scalar Registers*

![Figure 110. Vector-Scalar Register Elements](image)

<table>
<thead>
<tr>
<th>SD/UD/MD/DP 0</th>
<th>SD/UD/MD/DP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW/UW/MW/SP 0</td>
<td>SW/UW/MW/SP 1</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>

*Figure 110. Vector-Scalar Register Elements*

7.2.1.1 Floating-Point Registers

Chapter 4. Floating-Point Facility [Category: Floating-Point] provides 32 64-bit FPRs. Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point] also employs FPRs in decimal floating-point (DFP) operations. When VSX is implemented, the 32 FPRs are mapped to doubleword 0 of VSRs 0-31. For example, FPR[0] is located in doubleword element 0 of VSR[0], FPR[1] is located in doubleword element 0 of VSR[1], and so forth.

All instructions that operate on an FPR are redefined to operate on doubleword element 0 of the corresponding VSR. The contents of doubleword element 1 of the VSR corresponding to a source FPR or FPR pair for these instructions are ignored and the contents of doubleword element 1 of the VSR corresponding to the target FPR or FPR pair for these instructions are undefined.
### Figure 111. Floating-Point Registers as part of VSRs

<table>
<thead>
<tr>
<th>VSR[0]</th>
<th>FPR[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR[1]</td>
<td>FPR[1]</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>VSR[31]</td>
<td>FPR[31]</td>
</tr>
<tr>
<td>VSR[32]</td>
<td></td>
</tr>
<tr>
<td>VSR[33]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>VSR[62]</td>
<td></td>
</tr>
<tr>
<td>VSR[63]</td>
<td></td>
</tr>
</tbody>
</table>

0  | 63 | 127
7.2.1.2 Vector Registers

Chapter 5. Vector Facility [Category: Vector] provides 32 128-bit VRs. When VSX is implemented, the 32 VRs are mapped to VSRs 32-63. For example, VR[0] is located in VSR[32], VR[1] is located in VSR[33], and so forth. All instructions that operate on a VR are redefined to operate on the corresponding VSR.

Figure 112. Vector Registers as part of VSRs
### 7.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:19 and 32:55 are status bits. Bits 56:63 are control bits.

The exception status bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an `mcrfs`, `mtfsfi`, `mtfsf`, or `mtfsb0` instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be "exception status bits", and only FX is sticky.

#### Programming Note

Access to `Move To FPSCR` and `Move From FPSCR` instructions requires FP=1.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

The bit definitions for the FPSCR are as follows.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:28</td>
<td>Decimal Floating-Point Rounding Control (DRN)</td>
</tr>
<tr>
<td>32</td>
<td>Floating-Point Exception Summary (FX)</td>
</tr>
<tr>
<td>33</td>
<td>Floating-Point Enabled Exception Summary (FEX)</td>
</tr>
</tbody>
</table>

#### Programming Note

FX is defined not to be altered implicitly by `mtfsfi` and `mtfsf` because permitting these instructions to alter FX implicitly can cause a paradox. An example is an `mtfsfi` or `mtfsf` instruction that supplies 0 for FX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.

34 | Floating-Point Invalid Operation Exception Summary (VX) |
---|---|
| This bit is the OR of all the Invalid Operation exception bits. `mcrfs`, `mtfsfi`, `mtfsf`, and `mtfsb0` cannot alter VX explicitly. |

35 | Floating-Point Overflow Exception (OX) |
---|---|
| This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic, VSX Vector Floating-Point Arithmetic, VSX Scalar DP-SP Conversion class instruction causes an Overflow exception. See Section 7.4.3, “Floating-Point Overflow Exception” on page 351. |
| This bit can be set to 0 or 1 by a `Move To FPSCR` class instruction. |

36 | Floating-Point Underflow Exception (UX) |
---|---|
| This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic, VSX Vector Floating-Point Arithmetic, VSX Scalar DP-SP Conversion class instruction causes an Underflow exception. See Section 7.4.4, “Floating-Point Underflow Exception” on page 353. |
| This bit can be set to 0 or 1 by a `Move To FPSCR` class instruction. |

37 | Floating-Point Zero Divide Exception (ZX) |
---|---|
| This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic or VSX Vector Floating-Point Arithmetic class instruction causes a Zero Divide exception. See Section 7.4.2, “Floating-Point Zero Divide Exception” on page 349. |
| This bit can be set to 0 or 1 by a `Move To FPSCR` class instruction. |

38 | Floating-Point Inexact Exception (XX) |
---|---|
| This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic, VSX Vector Floating-Point Arithmetic, VSX Scalar Integer Conversion, VSX Vector Integer Conversion, VSX Scalar Round to Floating-Point Integer, or VSX Vector Round to Floating-Point Integer class instruction causes an Inexact exception. See Section 7.4.5, “Floating-Point Inexact Exception” on page 356. |
| This bit can be set to 0 or 1 by a `Move To FPSCR` class instruction. |
### Bits Definition

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>Floating-Point Invalid Operation Exception (SNAN) (VXSnan)</td>
</tr>
<tr>
<td>40</td>
<td>Floating-Point Invalid Operation Exception (Inf-Inf) (VXISI)</td>
</tr>
<tr>
<td>41</td>
<td>Floating-Point Invalid Operation Exception (Inf÷Inf) (VXIDI)</td>
</tr>
<tr>
<td>42</td>
<td>Floating-Point Invalid Operation Exception (Zero÷Zero) (VXZDZ)</td>
</tr>
<tr>
<td>43</td>
<td>Floating-Point Invalid Operation Exception (Inf×Zero) (VXIMZ)</td>
</tr>
<tr>
<td>44</td>
<td>Floating-Point Invalid Operation Exception (Invalid Compare) (VXVC)</td>
</tr>
<tr>
<td>45</td>
<td>Floating-Point Fraction Rounded (FR)</td>
</tr>
<tr>
<td>46</td>
<td>Floating-Point Fraction Inexact (FI)</td>
</tr>
</tbody>
</table>

**Bits 39**: This bit is set to 1 when a VSX Scalar Floating-Point and VSX Vector Floating-Point class instruction causes an SNaN type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

**Bits 40**: This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes an Infinity – Infinity type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

**Bits 41**: This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes an Infinity ÷ Infinity type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

**Bits 42**: This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes a Zero ÷ Zero type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

**Bits 43**: This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic and VSX Vector Floating-Point Arithmetic class instruction causes an Infinity × Zero type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

**Bits 44**: This bit is set to 1 when a VSX Scalar Compare Double-Precision, VSX Vector Compare Double-Precision, or VSX Vector Compare Single-Precision class instruction causes an Invalid Compare type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

**Bits 45**: This bit is set to 0 or 1 by VSX Scalar Floating-Point Arithmetic, VSX Scalar Integer Conversion, and VSX Scalar Round to Floating-Point Integer class instructions to indicate whether or not the fraction was incremented during rounding. See Section 7.3.2.6, “Rounding” on page 335. This bit is not sticky.

**Bits 46**: This bit is set to 0 or 1 by VSX Scalar Floating-Point Arithmetic, VSX Scalar Integer Conversion, and VSX Scalar Round to Floating-Point Integer class instructions to indicate whether or not the rounded result is inexact or the instruction caused a disabled Overflow exception. See Section 7.3.2.6 on page 335. This bit is not sticky.

See the definition of XX, above, regarding the relationship between FI and XX.
Bits Definition

47:51 Floating-Point Result Flags (FPRF) VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion, VSX Scalar Convert Integer to Double-Precision, and VSX Scalar Round to Double-Precision Integer class instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

For VSX Scalar Convert Double-Precision to Integer class instructions, the value placed into FPRF is undefined.

Additional details are as follows.

47 Floating-Point Result Class Descriptor (C) VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion, VSX Scalar Convert Integer to Double-Precision, and VSX Scalar Round to Double-Precision Integer class instructions set this bit with the FPCC bits, to indicate the class of the result as shown in Table 2, “Floating-Point Result Flags,” on page 327.

48:51 Floating-Point Condition Code (FPCC) VSX Scalar Compare Double-Precision instruction sets one of the FPCC bits to 1 and the other three FPCC bits to 0 based on the relative values of the operands being compared.

VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion, VSX Scalar Convert Integer to Double-Precision, and VSX Scalar Round to Double-Precision Integer class instructions set the FPCC bits with the C bit, to indicate the class of the result as shown in Table 2, “Floating-Point Result Flags,” on page 327. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 Floating-Point Less Than or Negative (FL)

49 Floating-Point Greater Than or Positive (FG)

50 Floating-Point Equal or Zero (FE)

51 Floating-Point Unordered or NaN (FU)

Bits Definition

52 Reserved

53 Floating-Point Invalid Operation Exception (Software-Defined Condition) (VXSOFT) This bit can be altered only by mcrfs, mtfsfi, mtfsf, mtfsb0, or mtfsb1. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

VXSOFT can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

Floating-Point Invalid Operation Exception (Invalid Square Root) (VXSQRT) This bit is set to 1 when a VSX Scalar Floating-Point Arithmetic or VSX Vector Floating-Point Arithmetic class instruction causes a Invalid Square Root type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

Floating-Point Invalid Operation Exception (Invalid Integer Convert) (VXCVI) This bit is set to 1 when a VSX Scalar Convert Double-Precision to Integer, VSX Vector Convert Double-Precision to Integer, or VSX Vector Convert Single-Precision to Integer class instruction causes a Invalid Integer Convert type Invalid Operation exception. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.

This bit can be set to 0 or 1 by a Move To FPSCR class instruction.

Floating-Point Invalid Operation Exception Enable (VE) This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Invalid Operation exceptions. See Section 7.4.1, “Floating-Point Invalid Operation Exception” on page 343.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>Floating-Point Overflow Exception Enable (OE)</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Overflow exceptions. See Section 7.4.3, &quot;Floating-Point Overflow Exception&quot; on page 351.</td>
</tr>
<tr>
<td>58</td>
<td>Floating-Point Underflow Exception Enable (UE)</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Underflow exceptions. See Section 7.4.4, &quot;Floating-Point Underflow Exception&quot; on page 353.</td>
</tr>
<tr>
<td>59</td>
<td>Floating-Point Zero Divide Exception Enable (ZE)</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Zero Divide exceptions. See Section 7.4.2, &quot;Floating-Point Zero Divide Exception&quot; on page 349.</td>
</tr>
<tr>
<td>60</td>
<td>Floating-Point Inexact Exception Enable (XE)</td>
</tr>
<tr>
<td></td>
<td>This bit is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions to enable trapping on Inexact exceptions. See Section 7.4.5, &quot;Floating-Point Inexact Exception&quot; on page 356.</td>
</tr>
<tr>
<td>61</td>
<td>Floating-Point Non-IEEE Mode (NI)</td>
</tr>
<tr>
<td></td>
<td>Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply. If floating-point non-IEEE mode is implemented, this bit has the following meaning.</td>
</tr>
<tr>
<td></td>
<td>0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard).</td>
</tr>
<tr>
<td></td>
<td>1 The processor is in floating-point non-IEEE mode.</td>
</tr>
<tr>
<td>61</td>
<td>Floating-Point Non-IEEE Mode (NI) (continued)</td>
</tr>
<tr>
<td></td>
<td>When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits is permitted to have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with NI=1, and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode is permitted to vary between implementations, and between different executions on the same implementation.</td>
</tr>
<tr>
<td>62:63</td>
<td>Floating-Point Rounding Control (RN)</td>
</tr>
<tr>
<td></td>
<td>This field is used by VSX Scalar Floating-Point and VSX Vector Floating-Point class instructions that round their result and the rounding mode is not implied by the opcode. This bit can be explicitly set or reset by a new Move To FPSCR class instruction. See Section 7.3.2.6, “Rounding” on page 335.</td>
</tr>
<tr>
<td></td>
<td>00 Round to Nearest Even</td>
</tr>
<tr>
<td></td>
<td>01 Round toward Zero</td>
</tr>
<tr>
<td></td>
<td>10 Round toward +Infinity</td>
</tr>
<tr>
<td></td>
<td>11 Round toward -Infinity</td>
</tr>
</tbody>
</table>

Programming Note

When the processor is in floating-point non-IEEE mode, the results of floating-point operations is permitted to be approximate, and performance for these operations might be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation is permitted to return 0 instead of a denormalized number and return a large number instead of an infinity.
### Table 2. Floating-Point Result Flags

<table>
<thead>
<tr>
<th>Result Flags</th>
<th>Result Value Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>– Infinity</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>– Normalized Number</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>– Denormalized Number</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>– Zero</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>+ Zero</td>
</tr>
<tr>
<td>1 0 1 0 0</td>
<td>+ Denormalized Number</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>+ Normalized Number</td>
</tr>
<tr>
<td>0 0 1 0 1</td>
<td>+ Infinity</td>
</tr>
</tbody>
</table>
7.3 VSX Operations

7.3.1 VSX Floating-Point Arithmetic Overview

This section describes the floating-point arithmetic and exception model supported by category Vector-Scalar Extension. Except for extensions to support 32-bit single-precision floating-point vector operations, the models are identical to that described in Chapter 4. Floating-Point Facility [Category: Floating-Point].

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic (hereafter referred to as the IEEE standard). That standard defines certain required "operations" (addition, subtraction, and so on). Herein, the term, floating-point operation, is used to refer to one of these required operations and to additional operations defined (e.g., those performed by Multiply-Add or Reciprocal Estimate instructions). A Non-IEEE mode is also provided. This mode, which is permitted to produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in VSRs; to move floating-point data between storage and these registers.

These instructions are divided into two categories.

- computational instructions
  
The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. There are two forms of computational instructions, scalar, which perform a single floating-point operation, and vector, which perform either two double-precision floating-point operations or four single-precision operations. Computational instructions place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 7.6.1.3 through 7.6.1.72.

- noncomputational instructions
  
The noncomputational instructions are those that perform loads and stores, move the contents of a VSR to another floating-point register possibly altering the sign, and select the value from one of two VSRs based on the value in a third VSR. The operations performed by these instructions are not considered floating-point operations. These instructions do not alter the Floating-Point Status and Control Register. They are the instructions listed in Sections 7.6.1.1, 7.6.1.2.1, and 7.6.1.8 through 7.6.1.9.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{\text{exponent}}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values that are “Not a Number” (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. NaNs might be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to categories Vector-Scalar Extension and Floating-Point: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the FPSCR. They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

- Invalid Operation exception (VX)
  
  - SNaN (VXSNAN)
  - Infinity–Infinity (VXSI)
  - Infinity±Infinity (VXIDI)
  - Zero±Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)
  - Invalid Integer Convert (VXCVI)
  
- Zero Divide exception (ZX)

- Overflow exception (OX)

- Underflow exception (UX)

- Inexact exception (XX)

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 323 for a description of these exception and enable bits, and Section 7.3.3, “VSX Floating-Point Execution Models” on page 337 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.
7.3.2 VSX Floating-Point Data

7.3.2.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats, 32-bit single-precision format and 64-bit double-precision format. The single-precision format is used for SP data in storage and registers. The double-precision format is used for DP data in storage and registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single-precision and double-precision formats is shown below.

Values in floating-point format are composed of three fields:

- **S**: sign bit
- **EXP**: exponent + bias
- **FRACTION**: fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (that is, the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Table 3.

---

### Table 3. IEEE floating-point fields

<table>
<thead>
<tr>
<th></th>
<th>Single-Precision Format</th>
<th>Double-Precision Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent Bias</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>Maximum Exponent (Emax)</td>
<td>+127</td>
<td>+1023</td>
</tr>
<tr>
<td>Minimum Exponent (Emin)</td>
<td>−126</td>
<td>−1022</td>
</tr>
<tr>
<td>Widths (bits):Format</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Sign</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Exponent</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>Fraction</td>
<td>23</td>
<td>52</td>
</tr>
<tr>
<td>Significand</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>Nmax</td>
<td>((1-2^{-24}) \times 2^{128} \approx 3.4 \times 10^{38})</td>
<td>((1-2^{-53}) \times 2^{1024} \approx 1.8 \times 10^{308})</td>
</tr>
<tr>
<td>Nmin</td>
<td>(1.0 \times 2^{-126} \approx 1.2 \times 10^{-38})</td>
<td>(1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308})</td>
</tr>
<tr>
<td>Dmin</td>
<td>(1.0 \times 2^{-149} \approx 1.4 \times 10^{-45})</td>
<td>(1.0 \times 2^{-1074} \approx 4.9 \times 10^{-324})</td>
</tr>
</tbody>
</table>

\(\approx\) Value is approximate.

- Dmin: Smallest (in magnitude) representable denormalized number.
- Nmax: Largest (in magnitude) representable number.
- Nmin: Smallest (in magnitude) representable normalized number.
7.3.2.2 Value Representation

This architecture defines numeric and nonnumeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 115.

Figure 115. Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)
These are values that have a biased exponent value in the range:

1 to 254 in single-precision format
1 to 2046 in double-precision format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

\[ \text{NOR} = (-1)^s \times 2^E \times (1.\text{fraction}) \]

where \( s \) is the sign, \( E \) is the unbiased exponent, and \( 1.\text{fraction} \) is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

Zero values (±0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to -0).

Denormalized numbers (±DEN)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

\[ \text{DEN} = (-1)^s \times 2^{\text{Emin}} \times (0.\text{fraction}) \]

where \( \text{Emin} \) is the minimum representable exponent value (−126 for single-precision, −1022 for double-precision).

Infinities (±INF)
These are values that have the maximum biased exponent value:

255 in single-precision format
2047 in double-precision format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

\[-\text{Infinity} < \text{every finite number} < +\text{Infinity}\]

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 7.4.1, "Floating-Point Invalid Operation Exception" on page 343.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

Not a Numbers (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0, the NaN is a Signaling NaN; otherwise it is a Quiet NaN.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.
Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation exception is disabled (VE=0). Quiet NaNs propagate through all floating-point operations except ordered comparison and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

Assume the following generic arithmetic templates.

\[
\begin{align*}
&f(src1, src3, src2) \\
&\text{ex: result} = (src1 \times src3) - src2 \\
&f(src1, src2) \\
&\text{ex: result} = src1 \times src2 \\
&\text{ex: result} = src1 + src2 \\
&f(src1) \\
&\text{ex: result} = f(src1)
\end{align*}
\]

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a trap-disabled Invalid Operation exception, the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

\[
\begin{align*}
&\text{if src1 is a NaN} \\
&\quad \text{then result} = \text{Quiet}(src1) \\
&\text{else if src2 is a NaN (if there is a src2)} \\
&\quad \text{then result} = \text{Quiet}(src2) \\
&\text{else if src3 is a NaN (if there is a src3)} \\
&\quad \text{then result} = \text{Quiet}(src3) \\
&\text{else if disabled invalid operation exception} \\
&\quad \text{then result} = \text{generated QNaN}
\end{align*}
\]

where Quiet(x) means x if x is a QNaN and x converted to a QNaN if x is an SNaN. Any instruction that generates a QNaN as the result of a disabled Invalid Operation exception generates the value 0x7FF8_0000_0000_0000 for double-precision and 0x7FC0_0000 for single-precision.

Note that the M-form multiply-add-type instructions use the B source operand to specify src3 and the T target operand to specify src2, whereas A-form multiply-add-type instructions use the B source operand to specify src2 and the T target operand to specify src3.

A double-precision NaN is considered to be representable in single-precision format if and only if the low-order 29 bits of the double-precision NaN’s fraction are zero.

### 7.3.2.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same signs, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation \(x - y\) is the same as the sign of the result of the add operation \(x + (-y)\).

When the sum of two operands with opposite sign, or the difference of two operands with the same signs, is exactly zero, the sign of the result is positive in all rounding modes except Round toward –Infinity, in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

- The sign of the result of a Square Root or Reciprocal Square Root Estimate operation is always positive, except that the square root of \(-0\) is \(-0\) and the reciprocal square root of \(-0\) is \(-\infty\).

- The sign of the result of a Convert From Integer or Round to Floating-Point Integer operation is the sign of the operand being converted.

For the Multiply-Add instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

### 7.3.2.4 Normalization and Denormalization

The intermediate result of an arithmetic instruction can require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the
significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 337) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result can have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 7.4.4, “Floating-Point Underflow Exception” on page 353. These rules can require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format’s minimum value. If any significant bits are lost in this shifting process, “Loss of Accuracy” has occurred (See Section 7.4.4, “Floating-Point Underflow Exception” on page 353) and Underflow exception is signaled.

Engineering Note
When denormalized numbers are operands of multiply, divide, and square root operations, some implementations might prenormalize the operands internally before performing the operations.

7.3.2.5 Data Handling and Precision

Scalar double-precision floating-point data is represented in double-precision format in VSRs and storage.

Vector double-precision floating-point data is represented in double-precision format in VSRs and storage.

Scalar single-precision floating-point data is represented in double-precision format in VSRs and single-precision format in storage.

Vector single-precision floating-point data is represented in single-precision format in VSRs and storage.

Double-precision operands may be used as input for double-precision scalar arithmetic operations.

Double-precision operands may be used as input for single-precision scalar arithmetic operations when trapping on overflow and underflow exceptions is disabled.

Single-precision operands may be used as input for double-precision and single-precision scalar arithmetic operations.

Double-precision operands may be used as input for double-precision vector arithmetic operations.

Single-precision operands may be used as input for single-precision vector arithmetic operations.

Instructions are also provided for manipulations which do not require double-precision or single-precision. In addition, instructions are provided to access an integer representation in GPRs.

Single-Precision Operands

For single-precision scalar data, a conversion from single-precision format to double-precision format is performed when loading from storage into a VSR and a conversion from double-precision format to single-precision format is performed when storing from a VSR to storage. No floating-point exceptions are caused by these instructions.

Instructions are provided to convert between single-precision and double-precision formats for scalar and vector data in VSRs.

An instruction is provided to explicitly convert a double format operand in a VSR to single-precision. Scalar single-precision floating-point is enabled with six types of instruction.

1. Load Scalar Single-Precision
   This form of instruction accesses a floating-point operand in single-precision format in storage, converts it to double-precision format, and loads it into a VSR. No floating-point exceptions are caused by these instructions.

2. Scalar Round to Single-Precision
   \texttt{xrsps} rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into a VSR in double-precision format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of
**xsresp, xsrsp** does not alter the value. Values greater in magnitude than $2^{319}$ when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than $2^{-318}$ when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled back into the normalized range.

3. **Scalar Convert Single-Precision to Double-Precision**

*xscvdpdp* accesses a floating-point operand in single-precision format from word element 0 of the source VSR, converts it to double-precision format, and places it into doubleword element 0 of the target VSR.

4. **Scalar Convert Double-Precision to Single-Precision**

*xscvdpsp* rounds the double-precision floating-point value in doubleword element 0 of the source VSR to single-precision, and places the result into word element 0 of the target VSR in single-precision format. This function would be used to port scalar floating-point data to a format compatible for single-precision vector operations. Values greater in magnitude than $2^{319}$ when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than $2^{-318}$ when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled back into the normalized range.

5. **VSX Scalar Single-Precision Arithmetic**

This form of instruction takes operands from the VSRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single-precision format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then placed into the target VSR in double-precision format. The result lies in the range supported by the single format.

If any input value is not representable in single-precision format and either OE=1 or UE=1, the result placed into the target VSR and the setting of status bits in the FPSCR are undefined.

For *xsresp* or *xsrqrtresp*, if the input value is finite and has an unbiased exponent greater than $+127$, the input value is interpreted as an Infinity.

6. **Store VSX Scalar Single-Precision**

*stxspx* converts a single-precision value that is in double-precision format to single-precision format and stores that operand into storage. No floating-point exceptions are caused by *stxspx*. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding five types.)

When the result of a *Load VSX Scalar Single-Precision* (*lxsspx*), a *VSX Scalar Round to Single-Precision* (*xsrsp*), or a *VSX Scalar Single-Precision Arithmetic* instruction is stored in a VSR, the low-order 29 bits of FRACTION are zero.

---

**Programming Note**

VSX Scalar Round to Single-Precision (*xsrsp*) is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. *xsrsp* should be used to convert double-precision floating-point values to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by an *xsrsp*.

---

1. VSX Scalar Single-Precision Arithmetic instructions:
   * xsaddsp, xsdivsp, xsmulsp, xsresp, xssubsp, xsmdasp, xsmaddsp, xsmsubasp, xsmsubbsp, xsnmaddsp, xsnmaddssp, xsnmmsubasp, xsnmmsubbsp
Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and integer processing, instructions are provided to convert between floating-point double and single-precision format and integer word and doubleword format in a VSR. Computation on integer-valued operands can be performed using arithmetic instructions of the required precision. (The results might not be integer values.) The three groups of instructions provided specifically to support integer-valued operands are described below.

1. Rounding to a floating-point integer

   a. VSX Scalar Round to Double-Precision Integer
      - VSX Scalar Round to Double-Precision Integer instructions round a double-precision operand to an integer value in double-precision format.
      - VSX Vector Round to Double-Precision Integer instructions round each double-precision vector operand element to an integer value in double-precision format.

   b. VSX Vector Round to Single-Precision Integer
      - VSX Vector Round to Single-Precision Integer instructions round each single-precision vector operand element to an integer value in single-precision format.

   c. VSX Round to Floating-Point Integer
      - VSX Round to Floating-Point Integer instructions can cause Invalid Operation (VXSNAN) exceptions.

   d. VSX Scalar Double-Precision to Integer Format Conversion
      - VSX Scalar Double-Precision to Integer Format Conversion instructions convert a double-precision operand to 32-bit or 64-bit signed or unsigned integer format.

   e. VSX Vector Double-Precision to Integer Format Conversion
      - VSX Vector Double-Precision to Integer Format Conversion instructions convert either double-precision or single-precision vector operand elements to 32-bit or 64-bit signed or unsigned integer format.

   f. VSX Vector Single-Precision to Integer Word Format Conversion
      - VSX Vector Single-Precision to Integer Word Format Conversion instructions convert the single-precision value in each odd-numbered word element of the source vector operand to a 64-bit signed or unsigned integer format.

   g. VSX Vector Single-Precision to Integer Format Conversion
      - VSX Vector Single-Precision to Integer Format Conversion instructions convert the single-precision value in each word element of the source vector operand to either a 32-bit signed or unsigned integer format.

1. VSX Scalar Round to Double-Precision Integer instructions:
   - xsrds, xsrdps, xsrdpm, xsrdpi, xsrdp
2. VSX Vector Round to Double-Precision Integer instructions:
   - xvrdps, xvrdpp, xvrdpm, xvrdpi, xvrdp
3. VSX Vector Round to Single-Precision Integer instructions:
   - xvrsii, xvrsipi, xvrsipim, xvrsipi, xvrszip
4. VSX Round to Floating-Point Integer instructions:
   - xsrdpi, xsrds, xsrdpm, xsrdpi, xsrds, xsrdpi, xvrdps, xvrdpp, xvrdpm, xvrdpi, xvrdp
5. VSX Scalar Double-Precision to Integer Format Conversion instructions:
   - xsuvdpsxds, xsvdpxsws, xsvdpxuds, xsuvdpxws
6. VSX Vector Double-Precision to Integer Format Conversion instructions:
   - xecvpsxws, xecvdpxws, xevdpxuds, xecvpsxws, xevdpxws
7. VSX Vector Single-Precision to Integer Doubleword Format Conversion instructions:
   - xevpspxws, xevpsxws
8. VSX Vector Single-Precision to Integer Word Format Conversion instructions:
   - xevpspxws, xevpsxws
Rounding is performed using Round Towards Zero rounding mode. These instructions can cause Invalid Operation (VXSNAN, VXCVI) and Inexact exceptions.

3. Converting integer format to floating-point format

VSX Scalar Integer Doubleword to Double-Precision Format Conversion\[^{[1]}\] instructions convert a 64-bit signed or unsigned integer to a double-precision floating-point value and returns the result in double-precision format.

VSX Scalar Integer Doubleword to Single-Precision Format Conversion\[^{[2]}\] instructions converts a 64-bit signed or unsigned integer to a single-precision floating-point value and returns the result in double-precision format.

VSX Vector Integer Doubleword to Double-Precision Format Conversion\[^{[3]}\] instructions converts the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to double-precision floating-point format.

VSX Vector Integer Doubleword to Single-Precision Format Conversion\[^{[4]}\] instructions convert the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to single-precision floating-point format.

VSX Vector Integer Word to Double-Precision Format Conversion\[^{[5]}\] instructions converts the 32-bit signed or unsigned integer in each odd-numbered word element in the source vector operand to double-precision floating-point format.

VSX Vector Integer Word to Single-Precision Format Conversion\[^{[6]}\] instructions convert the 32-bit signed or unsigned integer in each word element in the source vector operand to single-precision floating-point format.

Rounding is performed using the rounding mode specified in RN. Because of the limitations of the source format, only an Inexact exception can be generated.

7.3.2.6 Rounding

The material in this section applies to operations that have numeric operands (that is, operands that are not infinities or NaNs). Rounding the intermediate result of such an operation can cause an Overflow exception, an Underflow exception, or an Inexact exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 7.3.2.2, “Value Representation” and Section 7.4, “VSX Floating-Point Exceptions” for the cases not covered here.

The floating-point arithmetic, and rounding and conversion instructions round their intermediate results. With the exception of the estimate instructions, these instructions produce an intermediate result that can be regarded as having unbounded precision and exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target element of the target VSR in either double or single-precision format.

The scalar round to double-precision integer, vector round to double-precision integer, and convert double-precision to integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for round to double-precision integer instructions is normalized and put in double-precision format, and, for the convert double-precision to integer instructions, is converted to a signed or unsigned integer.

The vector round to single-precision integer and vector convert single-precision to integer instructions with biased exponents ranging from 126 through 178 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 179. (Intermediate results with biased exponents 179 or larger are already integers, and with biased exponents 125 or less round to zero.) After rounding, the final result for vector round to single-precision integer is normalized and put in double-precision format, and for
vector convert single-precision to integer is converted to a signed or unsigned integer.

FR and FI generally indicate the results of rounding. Each of the scalar instructions which rounds its intermediate result sets these bits. There are no vector instructions that modify FR and FI. If the fraction is incremented during rounding, FR is set to 1, otherwise FR is set to 0. If the result is inexact, FI is set to 1, otherwise FI is set to zero. The scalar round to double-precision integer instructions are exceptions to this rule, setting FR and FI to 0. The scalar double-precision estimate instructions set FR and FI to undefined values. The remaining scalar floating-point instructions do not alter FR and FI.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 323. These are encoded as follows.

<table>
<thead>
<tr>
<th>RN</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Round to Nearest Even</td>
</tr>
<tr>
<td>01</td>
<td>Round towards Zero</td>
</tr>
<tr>
<td>10</td>
<td>Round towards +Infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round towards -Infinity</td>
</tr>
</tbody>
</table>

A fifth rounding mode is provided in the round to floating-point integer instructions (Section 7.6.1.7.2 on page 368), Round to Nearest Away.

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 116 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes.

See Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 337 for a detailed explanation of rounding.

Figure 116 also summarizes the rounding actions for floating-point intermediate result for all supported rounding modes.

```
Round to Nearest Away
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is furthest away from 0.

Round to Nearest Even
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit is 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity
Choose Z1.

Round toward -Infinity
Choose Z2.
```

Figure 116.Selection of Z1 and Z2
7.3.3 VSX Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (that is, operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 7.3.2.2 and Section 7.3.3 for the cases not covered here.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow and underflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.
- Underflow during division using denormalized dividend and a large divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands.

VSX defines both scalar and vector double-precision floating-point operations to operate only on double-precision operands. VSX also defines vector single-precision floating-point operations to operate only on single-precision operands.

7.3.3.1 VSX Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

```
SCL  FRACTION  G  R  X
0 1 53 54 55
```

Figure 117. IEEE floating-point execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that appear to the low-order side of the R bit, resulting from either shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Table 4 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

<table>
<thead>
<tr>
<th>G</th>
<th>R</th>
<th>X</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IR is exact</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>IR closer to NL</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>IR midway between NL and NH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>IR closer to NH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Interpretation of G, R, and X bits

Table 5 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision operands.
floating-point numbers relative to the accumulator illustrated in Figure 117.

<table>
<thead>
<tr>
<th>Format</th>
<th>Guard</th>
<th>Round</th>
<th>Sticky</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>G bit</td>
<td>R bit</td>
<td>X bit</td>
</tr>
<tr>
<td>Single</td>
<td>24</td>
<td>25</td>
<td>OR of bits 26:52, G, R, X</td>
</tr>
</tbody>
</table>

Table 5. Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction.

Four user-selectable rounding modes are provided through RN as described in Section 7.3.2.6, “Rounding” on page 335. The rules for rounding in each mode are as follows.

– **Round to Nearest Even**
  
  **Guard bit = 0**
  
  The result is truncated.
  
  **Guard bit = 1**
  
  Depends on Round and Sticky bits:
  
  **Case a**
  
  If the Round or Sticky bit is 1 (inclusive), the result is incremented.
  
  **Case b**
  
  If the Round and Sticky bits are 0 (result midway between closest representable values), if the low-order bit of the result is 1, the result is incremented. Otherwise (the low-order bit of the result is 0), the result is truncated. This is the case of a tie rounded to even.

– **Round toward Zero**
  
  Choose the smaller in magnitude of Z₁ or Z₂. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.
  
  The result is truncated.

– **Round toward +Infinity**
  
  If positive, the result is incremented.
  
  If negative, the result is truncated.

– **Round toward −Infinity**
  
  If positive, the result is truncated.
  
  If negative, the result is incremented.

A fifth rounding mode is provided in the VSX Round to Floating-Point Integer instructions (Section 7.6.1.7.2 on page 368) with the rules for rounding as follows.

– **Round to Nearest Away**
  
  **Guard bit = 0**
  
  The result is truncated.
  
  **Guard bit = 1**
  
  The result is incremented.

If any of the Guard, Round, or Sticky bits is nonzero, the result is also inexact.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. Fraction bits are stored to the target VSR.

### 7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions

This architecture provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder.

32-bit arithmetic is similar, except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

![Multiply-add 64-bit execution model](image)

Figure 118. Multiply-add 64-bit execution model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input’s exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X’ bit. The add
operation also produces a result conforming to the above model with the $X'$ bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the $X'$ bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 6 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

<table>
<thead>
<tr>
<th>Format</th>
<th>Guard</th>
<th>Round</th>
<th>Sticky</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>53</td>
<td>54</td>
<td>OR of 55:105, $X'$</td>
</tr>
<tr>
<td>Single</td>
<td>24</td>
<td>25</td>
<td>OR of 26:105, $X'$</td>
</tr>
</tbody>
</table>

Table 6. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

The rules for rounding the intermediate result are the same as those given in Section 7.3.3.1.

If the instruction is a negative multiply-add or negative multiply-subtract type instruction, the final result is negated.
7.4 VSX Floating-Point Exceptions

This architecture defines the following floating-point exceptions under the IEEE-754 exception model:

- Invalid Operation exception
  - SNaN
  - Infinity ÷ Infinity
  - Infinity ÷ Zero
  - Infinity × Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide exception
- Overflow exception
- Underflow exception
- Inexact exception

These exceptions, other than Invalid Operation exception resulting from a Software-Defined Condition, can occur during execution of computational instructions. An Invalid Operation exception resulting from a Software-Defined Condition occurs when a Move To FPSCR instruction sets VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 341), whether and how the system floating-point enabled exception error handler is invoked. In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow exception depends on the setting of the enable bit.

A single instruction, other than mtfsfi or mtfsf, can set more than one exception bit only in the following cases:

- An Inexact exception can be set with an Overflow exception.
- An Inexact exception can be set with an Underflow exception.
- An Invalid Operation exception (SNaN) is set with an Invalid Operation exception (Infinity × 0) for multiply-add class instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Compare) for ordered comparison instructions.
- An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Integer Convert) for convert to integer instructions.

When an exception occurs, the writing of a result to the target register can be suppressed, or a result can be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the certain kinds of exceptions, based on whether the instruction is a vector or a scalar instruction, so that there is no possibility that one of the operands is lost. For other kinds of exceptions and also depending on whether the instruction is a vector or a scalar instruction, a result is generated and written to the destination specified by the instruction causing the exception. The result can be a different value for the enabled and disabled conditions for some of these exceptions. Table 7 lists the types of exceptions and indicates whether a result is written to the target VSR or suppressed.

<table>
<thead>
<tr>
<th>On exception type...</th>
<th>Scalar Instruction Results</th>
<th>Vector Instruction Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled Invalid Operation</td>
<td>suppressed</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Zero Divide</td>
<td>suppressed</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Overflow</td>
<td>written</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Underflow</td>
<td>written</td>
<td>suppressed</td>
</tr>
<tr>
<td>Enabled Inexact</td>
<td>written</td>
<td>suppressed</td>
</tr>
<tr>
<td>Disabled Invalid Operation</td>
<td>written</td>
<td>written</td>
</tr>
</tbody>
</table>

Table 7. Exception Types Result Suppression
The subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of traps and trap handlers. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the trap enabled case; the expectation is that the exception is detected by software, which revises the result. An FPSCR exception enable bit of 0 causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case. The expectation is that the exception is not detected by software, which uses the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is required for all exceptions, all FPSCR exception enable bits must be set to 0, and Ignore Exceptions Mode (see below) should be used. In this case, the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits, if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1, and a mode other than Ignore Exceptions Mode must be used. In this case, the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The system floating-point enabled exception error handler is also invoked if a Move To FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1. The Move To FPSCR instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception. The effects of the four possible settings of these bits are as follows.

### FE0 FE1 Description

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions Mode</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable Mode</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise Mode</td>
</tr>
</tbody>
</table>

Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.

The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction might have been used by or might have affected subsequent instructions that are executed before the error handler is invoked.

The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

The FE0 and FE1 bits must be defined in Book III in a manner such that they can be changed dynamically and can easily be treated as part of a process’ state.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system...
floating-point enabled exception error handler is invoked have been completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction, and there is only one such instruction. Otherwise, it has not begun execution, or has been partially executed in some cases, as described in Book III.

To obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode can degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

In any of the three non-Precise modes, a Floating-Point Status and Control Register instruction can be used to force any exceptions, because of instructions initiated before the Floating-Point Status and Control Register instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In both Imprecise modes, a Floating-Point Status and Control Register instruction can be used to force any invocations of the system floating-point enabled exception error handler that result from instructions initiated before the Floating-Point Status and Control Register instruction to occur. This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. It always applies in the latter case.

It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.
7.4.1 Floating-Point Invalid Operation Exception

7.4.1.1 Definition

An Invalid Operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

**SNaN**
Any floating-point operation on a Signaling NaN.

**Infinity–Infinity**
Magnitude subtraction of infinities.

**Infinity×Infinity**
Floating-point division of infinity by infinity.

**Zero÷Zero**
Floating-point division of zero by zero.

**Infinity × Zero**
Floating-point multiplication of infinity by zero.

**Invalid Compare**
Floating-point ordered comparison involving a NaN.

**Invalid Square Root**
Floating-point square root or reciprocal square root of a nonzero negative number.

**Invalid Integer Convert**
Floating-point-to-integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN.

An Invalid Operation exception also occurs when an *mtfsfi*, *mtfsf*, or *mtfsb1* instruction is executed that sets VXSOFT to 1 (Software-Defined Condition).

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

7.4.1.2 Action for VE=1

When Invalid Operation exception is enabled (VE=1) and an Invalid Operation exception occurs, the following actions are taken:

For *VSX Scalar Floating-Point Arithmetic, VSX Scalar DP-SP Conversion, VSX Scalar Convert Floating-Point to Integer, and VSX Scalar Round to Floating-Point Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXISI (if Infinity–Infinity)
   VXIDI (if Infinity×Infinity)
   VXZDZ (if Zero÷Zero)
   VXIMZ (if Infinity×Zero)
   VXSQRT (if Invalid Square Root)
   VXCVI (if Invalid Integer Convert)

2. Update of VSR[XT] is suppressed.

3. FR and FI are set to zero.

4. FPRF is unchanged.

For *VSX Scalar Floating-Point Compare* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXVC (if Invalid Compare)

2. FR, FI, and C are unchanged.

3. FPCC is set to reflect unordered.

For *VSX Vector Floating-Point Arithmetic, VSX Vector Floating-Point Compare, VSX Vector DP-SP Conversion, VSX Vector Convert Floating-Point to Integer, and VSX Vector Round to Floating-Point Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXISI (if Infinity–Infinity)
   VXIDI (if Infinity×Infinity)
   VXZDZ (if Zero÷Zero)
   VXIMZ (if Infinity×Zero)
   VXVC (if Invalid Compare)
   VXSQRT (if Invalid Square Root)
   VXCVI (if Invalid Integer Convert)

2. Update of VSR[XT] is suppressed for all vector elements.

3. FR and FI are unchanged.

4. FPRF is unchanged.
7.4.1.3 Action for VE=0

When Invalid Operation exception is disabled (VE=0) and an Invalid Operation exception occurs, the following actions are taken:

For the VSX Scalar round and Convert Double-Precision to Single-Precision format (xscvdpsp) instruction:

1. VXSNAN is set to 1.

2. The single-precision representation of a Quiet NaN is placed into word element 0 of VSR[XT]. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class of the result (Quiet NaN).

For the VSX Vector Single-Precision Arithmetic instructions, VSX Vector Single-Precision Maximum/Minimum instructions, the VSX Vector round and Convert Double-Precision to Single-Precision format (xvcvdpsp) instruction, and the VSX Vector Round to Single-Precision Integer instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXISI (if Infinity – Infinity)
   - VXIDI (if Infinity ÷ Infinity)
   - VXZDZ (if Zero ÷ Zero)
   - VXIMZ (if Infinity × Zero)
   - VXSQRT (if Invalid Square Root)

2. The single-precision representation of a Quiet NaN is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the VSX Scalar Double-Precision Arithmetic instructions, VSX Scalar Double-Precision Maximum/Minimum instructions, the VSX Scalar Convert Single-Precision to Double-Precision format (xscvsdpd) instruction, and the VSX Scalar Round to Double-Precision Integer instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXISI (if Infinity – Infinity)
   - VXIDI (if Infinity ÷ Infinity)
   - VXZDZ (if Zero ÷ Zero)
   - VXIMZ (if Infinity × Zero)
   - VXSQRT (if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class of the result (Quiet NaN).

For the VSX Vector Double-Precision Arithmetic instructions, VSX Vector Double-Precision Maximum/Minimum instructions, the VSX Vector Convert Single-Precision to Double-Precision format (xvcvsdpd) instruction, and the VSX Vector Round to Double-Precision Integer instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXISI (if Infinity – Infinity)
   - VXIDI (if Infinity ÷ Infinity)
   - VXZDZ (if Zero ÷ Zero)
   - VXIMZ (if Infinity × Zero)
   - VXSQRT (if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.
For the **VSX Scalar Convert Double-Precision to Signed Integer Doubleword** (**xscvdpsxd**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the **VSX Scalar Convert Double-Precision to Signed Integer Word** (**xscvdpsxw**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the **VSX Scalar Convert Double-Precision to Unsigned Integer Doubleword** (**xscvdpuxd**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the **VSX Scalar Convert Double-Precision to Unsigned Integer Word** (**xscvdpuxw**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN  (if SNaN)
   VXCVI   (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.
For the VSX Vector Convert Double-Precision to Signed Integer Doubleword (xvcvdpsxd) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. \(0x7FFF_FFFF_FFFF_FFFF\) is placed into doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a positive number or +Infinity.

   \(0x8000_0000_0000_0000\) is placed into its respective doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Double-Precision to Unsigned Integer Doubleword (xvcvdpuxd) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. \(0xFFFF_FFFF_FFFF_FFFF\) is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   \(0x0000_0000_0000_0000\) is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Double-Precision to Signed Integer Word (xvcvdpsxw) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. \(0x7FFF_FFFF\) is placed into word element \(i\times2\) of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   \(0x8000_0000\) is placed into word element \(i\times2+1\) of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element \(i\times2+1\) of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Double-Precision to Unsigned Integer Word (xvcvdpuxw) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. \(0xFFFF_FFFF\) is placed into word element \(i\times2\) of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   \(0x0000_0000\) is placed into word element \(i\times2\) of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element \(i\times2+1\) of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Single-Precision to Signed Integer Doubleword (xvcvspsxd) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. \(0x7FFF_FFFF_FFFF_FFFF\) is placed into doubleword element i of VSR[XT] if the single-precision operand in word element \(i\times2\) of VSR[XB] is a positive number or +Infinity.

   \(0x8000_0000_0000_0000\) is placed into doubleword element i of VSR[XT] if the single-precision operand in word element \(i\times2\) of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.
For the VSX Vector Convert Single-Precision to
Unsigned Integer Doubleword (**xvcvspuxd**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXCVI (if Invalid Integer Convert)

2. **0xFFFF_FFFF_FFFF_FFFF** is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.
   - **0x0000_0000_0000_0000** is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Single-Precision to
Signed Integer Word (**xvcvspsxw**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXCVI (if Invalid Integer Convert)

2. **0x7FFF_FFFF** is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a positive number or +Infinity.
   - **0x8000_0000** is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a negative number, -Infinity, or NaN.
   - The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Single-Precision to
Unsigned Integer Word (**xvcvspuxw**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXCVI (if Invalid Integer Convert)

2. **0xFFFF_FFFF** is placed into word element i of VSR[XT] if the single-precision operand in the corresponding word element i×2 of VSR[XB] is a positive number or +Infinity.
   - **0x0000_0000** is placed into word element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Vector Convert Single-Precision to
Signed Integer Word (**xvcvspsxw**) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXCVI (if Invalid Integer Convert)

2. **0x7FFFFFFF** is placed into word element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.
   - **0x8000_0000** is placed into word element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the VSX Scalar Floating-Point Compare instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXCVI (if Invalid Integer Convert)

2. **0x0000_0000** is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the VSX Vector Compare Single-Precision instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   - VXSNAN (if SNaN)
   - VXCVI (if Invalid Integer Convert)

2. **0x0000_0000** is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.
For the vector double-precision compare instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.
   
   \[\text{VXSNAN} \quad (\text{if SNaN})\]
   \[\text{VXCVI} \quad (\text{if Invalid Integer Convert})\]

2. 0x0000_0000_0000_0000 is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

7.4.2 Floating-Point Zero Divide Exception

7.4.2.1 Definition

A Zero Divide exception occurs when a VSX Floating-Point Divide instruction is executed with a zero divisor value and a finite nonzero dividend value. A Zero Divide exception also occurs when a VSX Floating-Point Reciprocal Estimate instruction or a VSX Floating-Point Reciprocal Square Root Estimate instruction is executed with an operand value of zero.

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

7.4.2.2 Action for ZE=1

When Zero Divide exception is enabled (ZE=1) and a Zero Divide exception occurs, the following actions are taken:

For VSX Scalar Floating-Point Divide instructions, VSX Scalar Floating-Point Reciprocal Estimate instructions, and VSX Scalar Floating-Point Reciprocal Square Root Estimate instructions, do the following.

1. ZX is set to 1.
2. Update of VSR[XT] is suppressed.
3. FR and FI are set to 0.
4. FPRF is unchanged.

For VSX Vector Floating-Point Divide instructions, VSX Vector Floating-Point Reciprocal Estimate instructions, and VSX Vector Floating-Point Reciprocal Square Root Estimate instructions, do the following.

1. ZX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR and FI are unchanged.
4. FPRF is unchanged.

1. VSX Floating-Point Divide instructions:
   - xsdivdp, xsdivsp, xvdivdp, xvdivsp
2. VSX Floating-Point Reciprocal Estimate instructions:
   - xsredp, xsresp, xvredp, xvresp
3. VSX Floating-Point Reciprocal Square Root Estimate instructions:
   - xsrsqrtedp, xsrsqrtesp, xvsqrtedp, xvsqrtresp
4. VSX Scalar Floating-Point Divide instructions:
   - xsdivdp, xsdivsp
5. VSX Scalar Floating-Point Reciprocal Estimate instructions:
   - xsredp, xsresp
6. VSX Scalar Floating-Point Reciprocal Square Root Estimate instructions:
   - xsrsqrtedp, xsrsqrtesp
7. VSX Vector Floating-Point Divide instructions:
   - xvdivdp, xvdivsp
8. VSX Vector Floating-Point Reciprocal Estimate instructions:
   - xvredp, xvresp
9. VSX Vector Floating-Point Reciprocal Square Root Estimate instructions:
   - xvsqrtedp, xvsqrtresp
7.4.2.3 Action for ZE=0

When Zero Divide exception is disabled (ZE=0) and a Zero Divide exception occurs, the following actions are taken:

For VSX Scalar Floating-Point Divide\[1\] instructions, do the following.

1. ZX is set to 1.
2. An Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class and sign of the result (± Infinity).

For VSX Vector Divide Double-Precision \(\text{xvdivdp}\), do the following.

1.ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For VSX Vector Divide Single-Precision \(\text{xvdivsp}\), do the following.

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

For VSX Scalar Floating-Point Reciprocal Estimate\[2\] instructions and VSX Scalar Floating-Point Reciprocal Square Root Estimate\[3\] instructions, do the following.

1. ZX is set to 1.
2. An Infinity, having the sign of the source operand, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class and sign of the result (± Infinity).

For the VSX Vector Reciprocal Estimate Double-Precision \(\text{xvredp}\) and VSX Vector Reciprocal Square Root Estimate Double-Precision \(\text{xvrsqrtedp}\) instructions:

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For the VSX Vector Reciprocal Estimate Single-Precision \(\text{xvresp}\) and VSX Vector Reciprocal Square Root Estimate Single-Precision \(\text{xvrsqrtesp}\) instructions:

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

1. VSX Scalar Floating-Point Divide instructions: \(\text{xsdvdp, xsdivsp}\)
2. VSX Scalar Floating-Point Reciprocal Estimate instructions: \(\text{xrsrdp, xsrsesp}\)
3. VSX Scalar Floating-Point Reciprocal Square Root Estimate instructions: \(\text{xrsqrtedp, xsrqsqrtresp}\)
7.4.3 Floating-Point Overflow Exception

7.4.3.1 Definition

An Overflow exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSR.

7.4.3.2 Action for OE=1

When Overflow exception is enabled (OE=1) and an Overflow exception occurs, the following actions are taken:

For the VSX Vector round and Convert Double-Precision to Single-Precision format (xscvdpsp) instruction:

1. OX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is less than or equal to 318 (Emax+192), the exponent is adjusted by subtracting 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For VSX Scalar Double-Precision Arithmetic\(^1\) instructions, do the following.

1. OX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by subtracting 1536.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For VSX Scalar Single-Precision Arithmetic\(^2\) instructions, do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 192.
3. The adjusted and rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For VSX Vector Double-Precision Arithmetic\(^3\) instructions, VSX Vector Single-Precision Arithmetic\(^4\) instructions, and VSX Vector round and Convert Double-Precision to Single-Precision format instruction (xvcvdpsp), do the following.

1. OX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR, FI, and FPRF are not modified.

---

1. VSX Scalar Double-Precision Arithmetic instructions:
   - xsadddp, xsdvdp, xsmuldp, xsredp, xssubdp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsnmaddadp, xsnmaddmdp, xsnmsubadp, xsnmsubmdp

2. VSX Scalar Single-Precision Arithmetic instructions:
   - xsaddsp, xsdivsp, xsmulsp, xssresp, xssubsp, xsmaddasp, xsmaddmsp, xsmsubasp, xsmsubmsp, xsnmaddasp, xsnmaddmsp, xsnmsubasp, xsnmsubmsp

3. VSX Vector Double-Precision Arithmetic instructions:
   - xvadddp, xvdvdp, xvmuldp, xvredp, xvsbmdp, xvmaddadp, xvmaddmdp, xvmsubadp, xvmsubmdp, xvmnmaddadp, xvmnmaddmdp, xvmnmsubadp, xvmnmsubmdp

4. VSX Vector Single-Precision Arithmetic instructions:
   - xvaddsp, xvdvsp, xvmulsp, xvresp, xvsbsp, xvmaddasp, xvmaddmsp, xvmsubasp, xvmsubmsp, xvmnmaddasp, xvmnmaddmsp, xvmnmsubasp, xvmnmsubmsp
7.4.3.3 Action for OE=0

When Overflow exception is disabled (OE=0) and an Overflow exception occurs, the following actions are taken:

1. OX and XX are set to 1.

2. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:

   **Round to Nearest Even**
   
   For negative overflow, the result is $-\infty$.
   
   For positive overflow, the result is $+\infty$.

   **Round toward Zero**
   
   For negative overflow, the result is the format's most negative finite number.
   
   For positive overflow, the result is the format's most positive finite number.

   **Round toward $+\infty$**
   
   For negative overflow, the result is the format's most negative finite number.
   
   For positive overflow, the result is $+\infty$.

   **Round toward $-\infty$**
   
   For negative overflow, the result is $-\infty$.
   
   For positive overflow, the result is the format's most positive finite number.

3. The result is placed into word element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.

4. FR is undefined.

5. Fi is set to 1.

6. FPRF is set to indicate the class and sign of the result.

For VSX Scalar Double-Precision Arithmetic\(^1\) instructions and VSX Scalar Single-Precision Arithmetic\(^2\) instructions, do the following.

3. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are undefined.

4. FR is undefined.

5. Fi is set to 1.

6. FPRF is set to indicate the class and sign of the result.

For VSX Vector Double-Precision Arithmetic\(^3\) instructions, do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

4. FR, Fi, and FPRF are not modified.

For VSX Vector Single-Precision Arithmetic\(^4\) instructions and VSX Vector round and Convert Double-Precision to Single-Precision format (\texttt{xvcvdpsp}), do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format.

4. FR, Fi, and FPRF are not modified.

---

1. VSX Scalar Double-Precision Arithmetic instructions:
   \texttt{xsadddp, xsdivdp, xsmuldp, xsmredp, xssubdp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsnaddddp, xsnaddmdp, xsnsubadp, xsnsubmdp}

2. VSX Scalar Single-Precision Arithmetic instructions:
   \texttt{xsaddsp, xsdivsp, xsmulsp, xsmresp, xsresp, xssubsp, xsmaddasp, xsmaddmsp, xsmsubasp, xsmsubmsp, xsnaddadsp, xsnaddmdsp, xsnsubadsp, xsnsubmdsp}

3. VSX Vector Double-Precision Arithmetic instructions:
   \texttt{xvadddp, xvdivdp, xvmuldp, xvredp, xsresvp, xsvsubvp, xvmaddadp, xvmaddmdp, xvmsubadp, xvmsubmdp, xvmnaddadp, xvmnaddmdp, xvmnsubadp, xvmnsubmdp}

4. VSX Vector Single-Precision Arithmetic instructions:
   \texttt{xvaddsp, xvdivsp, xvmulsp, xvresp, xsvsubsp, xvmaddasp, xvmaddmsp, xvmsubasp, xvmsubmsp, xvmnaddasp, xvmnaddmsp, xvmnsubasp, xvmnsubmsp}
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

7.4.4 Floating-Point Underflow Exception

7.4.4.1 Definition

Underflow exception is defined separately for the enabled and disabled states:

**Enabled:**
Underflow occurs when the intermediate result is "Tiny".

**Disabled:**
Underflow occurs when the intermediate result is "Tiny" and there is "Loss of Accuracy".

A tiny result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and Underflow exception is disabled (UE=0), the intermediate result is denormalized (see Section 7.3.2.4, "Normalization and Denormalization" on page 331) and rounded (see Section 7.3.2.6, "Rounding" on page 335) before being placed into the target VSR.

Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

7.4.4.2 Action for UE=1

When Underflow exception is enabled (UE=1) and an Underflow exception occurs, the following actions are taken:

For VSX Scalar round and Convert Double-Precision to Single-Precision format (xscvdpsp), do the following.
1. UX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is greater than or equal to -319 (Emin-192), the exponent is adjusted by adding 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For VSX Scalar Double-Precision Arithmetic\(^1\) instructions and VSX Scalar Double-Precision Reciprocal Estimate (xsredp), do the following.
1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 1536.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

---

\(^1\) VSX Scalar Double-Precision Arithmetic instructions: xsaddp, xsdivdp, xsmuld, xssubdp, xsmadddp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsnmadddp, xsnmaddmdp, xsnmsubadp, xsnmsubmdp
For VSX Scalar Single-Precision Arithmetic\(^1\) instructions and VSX Scalar Single-Precision Reciprocal Estimate (\texttt{xsresp})

1. UX is set to 1.

2. The exponent is adjusted by adding 192.

3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.

4. FPRF is set to indicate the class and sign of the result (\pm Normal Number).

### Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow exception, to simulate a "trap disabled" environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized and correctly rounded.

For VSX Vector Floating-Point Arithmetic\(^2\) instructions, VSX Vector Floating-Point Reciprocal Estimate\(^3\) instructions, and VSX Vector round and Convert Double-Precision to Single-Precision format (\texttt{xvcvdpsp}), do the following:

1. UX is set to 1.

2. Update of VSR[XT] is suppressed for all vector elements.

3. FR, FI, and FPRF are not modified.

### 7.4.4.3 Action for UE=0

When Underflow exception is disabled (UE=0) and an Underflow exception occurs, the following actions are taken:

For VSX Scalar round and Convert Double-Precision to Single-Precision format (\texttt{xscvdpsp})

1. UX is set to 1.

2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For VSX Scalar Floating-Point Arithmetic\(^4\) instructions and VSX Scalar Reciprocal Estimate\(^5\) instructions, do the following:

1. UX is set to 1.

2. The result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For VSX Vector Double-Precision Arithmetic\(^6\) instructions and VSX Vector Reciprocal Estimate Double-Precision (\texttt{xvredp})

1. UX is set to 1.

2. For each vector element causing an Underflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

3. FR, FI, and FPRF are not modified.

---

1. VSX Scalar Single-Precision Arithmetic instructions: \texttt{xsdssp}, \texttt{xsdvssp}, \texttt{xsmulp}, \texttt{xssubp}, \texttt{xsmaddsp}, \texttt{xsmaddmsp}, \texttt{xmsubasp}, \texttt{xmsubmsp}, \texttt{xsmaddasp}, \texttt{xsmaddmsp}, \texttt{xsnmsubasp}, \texttt{xsnmsubmsp}

2. VSX Vector Arithmetic instructions: \texttt{xvdssp}, \texttt{xvdvdp}, \texttt{xvmulp}, \texttt{xsvsubdp}, \texttt{xsvadsp}, \texttt{xsvdivsp}, \texttt{xsvmlsp}, \texttt{xsvsubp}, \texttt{xsvmaddsp}, \texttt{xsvmadddp}, \texttt{vxvsubdpp}, \texttt{vxvsubdmdp}, \texttt{vxvmaddsp}, \texttt{vxvmadddp}, \texttt{vxvmsubdp}, \texttt{vxvmsubdpp}, \texttt{vxvmsubasp}, \texttt{vxvmsubmsp}, \texttt{vxvmaddasp}, \texttt{vxvmadddmp}, \texttt{vxvmsubdsp}, \texttt{vxvmsubdmp}, \texttt{vxvmsubasp}, \texttt{vxvmsubmsp}

3. VSX Vector Floating-Point Reciprocal Estimate instructions: \texttt{xvredp}, \texttt{xvresp}

4. VSX Scalar Floating-Point Arithmetic instructions: \texttt{xvdssp}, \texttt{xvdvdp}, \texttt{xvmulp}, \texttt{xsvsubdp}, \texttt{xsvadsp}, \texttt{xsvdivsp}, \texttt{xsvmlsp}, \texttt{xsvsubp}, \texttt{xsvmaddsp}, \texttt{xsvmadddp}, \texttt{vxvsubdpp}, \texttt{vxvsubdmdp}, \texttt{vxvmaddsp}, \texttt{vxvmadddp}, \texttt{vxvmsubdp}, \texttt{vxvmsubdpp}, \texttt{vxvmsubasp}, \texttt{vxvmsubmsp}

5. VSX Scalar Reciprocal Estimate instructions: \texttt{xvredp}, \texttt{xvresp}

6. VSX Vector Double-Precision Arithmetic instructions: \texttt{xvdssp}, \texttt{xvdvdp}, \texttt{xvmulp}, \texttt{xsvsubdp}, \texttt{xsvadsp}, \texttt{xsvmaddsp}, \texttt{xsvmadddp}, \texttt{vxvsubdpp}, \texttt{vxvsubdmdp}, \texttt{vxvmaddsp}, \texttt{vxvmadddmp}, \texttt{vxvmsubdsp}, \texttt{vxvmsubdmp}, \texttt{vxvmsubasp}, \texttt{vxvmsubmsp}
For VSX Vector Single-Precision Arithmetic\(^1\) instructions, VSX Vector Reciprocal Estimate Single-Precision (\texttt{xvresp}), and VSX Vector round and Convert Double-Precision to Single-Precision format (\texttt{xvcvdpsp}), do the following.

1. UX is set to 1.

2. For each vector element causing an Underflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format.

3. FR, FI, and FPRF are not modified.

---

\(^1\) VSX Vector Single-Precision Arithmetic instructions:
\texttt{xvaddsp, xvdvsp, xvmlsp, xvsbasp, xvmaddsp, xvmaddmsp, xvmsubasp, xvmsubbasp, xvmaddasp, xvmaddmssp, xvmmsubasp, xvmmsubbasp}
7.4.5 Floating-Point Inexact Exception

7.4.5.1 Definition

An Inexact exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow exception or an enabled Underflow exception, an Inexact exception also occurs only if the significands of the rounded result and the intermediate result differ.)

2. The rounded result overflows and Overflow exception is disabled.

The action to be taken depends on the setting of the Inexact Exception Enable bit of the FPSCR.

7.4.5.2 Action for XE=1

When Inexact exception is enabled (UE=1) and an Inexact exception occurs, the following actions are taken:

For the VSX Vector round and Convert Double-Precision to Single-Precision format (xscvdpsp) instruction:

1. XX is set to 1.
2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For the VSX Scalar Floating-Point Arithmetic instructions:
- xsaddp, xsdivp, xsmulp, xssubp, xsmaddp, xsmaddsp, xsmsubp, xsmsubsp, xsnmaddp, xsnmaddsp, xsnmsubp, xsnmsubsp, xsnmadddp, xsnmaddsp, xsnmsubdp, xsnmsubsp, xsmaddmp, xsmaddsp, xsmsubmp, xsmsubsp, xsnmaddmp, xsnmaddsp, xsnmsubmp, xsnmsubsp

- For VSX Scalar Floating-Point Reciprocal Estimate instructions:
- xsvcpp, xsvcvmpp, xsvcppsp, xsvcvmppsp

- For VSX Vector Floating-Point Arithmetic instructions:
- xvadddp, xvdivdp, xvmpdp, xvsubdp, xsaddsp, xvdivsp, xvmulp, xvsubsp, xsmaddsp, xsmaddmp, xsmaddsp, xsmsubsp, xsnmaddsp, xsnmaddmp, xsnmsubsp, xsnmaddsp, xsnmaddmp, xsnmaddsp, xsnmsubsp, xsnmaddsp

1. VSX Scalar Floating-Point Arithmetic instructions:
   - xsaddp, xsdivp, xsmulp, xssubp, xsmaddp, xsmaddsp, xsmsubp, xsmsubsp, xsnmaddp, xsnmaddsp, xsnmsubp, xsnmsubsp, xsmaddmp, xsmaddsp, xsmsubmp, xsmsubsp, xsnmaddmp, xsnmaddsp, xsnmsubmp, xsnmsubsp

2. VSX Vector Integer to Floating-Point Format Conversion instructions:
   - xsvcpp, xsvcppsp, xsvcppmp, xsvcppmpsp

3. VSX Vector Floating-Point to Integer Word Format Conversion instructions:
   - xsvcpp, xsvcppws, xsvcppwsp

4. VSX Vector Floating-Point Arithmetic instructions:
   - xvaddp, xvdivp, xvmulp, xvsubp, xsaddp, xvdivp, xvmulp, xvsubp, xsmaddp, xsmaddsp, xsmsubp, xsnmaddp, xsnmaddsp, xsnmsubp, xsnmaddsp, xsnmaddmp, xsnmaddsp, xsnmsubmp, xsnmsubsp, xsnmaddsp, xsnmaddmp, xsnmaddsp, xsnmsubmp, xsnmsubsp
7.4.5.3 Action for XE=0

When Inexact exception is disabled (XE=0) and an Inexact exception occurs, the following actions are taken:

For **VSX Scalar round and Convert Double-Precision to Single-Precision format (xscvdpsp)**, do the following.

1. XX is set to 1.
2. The result is placed into word element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For **VSX Scalar Double-Precision Arithmetic** instructions, **VSX Scalar Single-Precision Arithmetic** instructions, **VSX Scalar Round to Single-Precision (xsrsp)**, the **VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode (xsrdpic)**, and **VSX Scalar Round to Double-Precision Format Conversion** instructions, do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For **VSX Scalar Convert Double-Precision To Integer Word format with Saturate** instructions, do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For **VSX Vector Double-Precision Arithmetic** instructions, do the following.

1. XX is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For **VSX Vector Single-Precision Arithmetic** instructions, do the following.

1. XX is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

5. **VSX Vector Floating-Point Reciprocal Estimate instructions**: xvredp, xvresp
6. **VSX Vector Double-Precision to Integer Format Conversion instructions**: xvcvdpsdxds, xvcvdpsxws, xvcvdpxdpsxds, xvcvdpxwxs
7. **VSX Vector Integer to Floating-Point Format Conversion instructions**: xvcvsxdp, xvcvsxdsp, xvcvsxdsdxsp, xvcvsxdsxwsp, xvcvsxwsp, xvcvxwsp
1. **VSX Scalar Double-Precision Arithmetic instructions**: xsaddp, xssubdp, xsmdulp, xsdivdp, xssqrtsp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp
2. **VSX Scalar Single-Precision Arithmetic instructions**: xsaddsp, xssubsp, xsmulp, xsdivsp, xssqrtsp, xsmaddsp, xsmaddmssp, xsmsubasp, xsmsubmsp, xsmaddasp, xsmaddmssp, xsmsubasp, xsmsubmsp
3. **VSX Scalar Integer to Double-Precision Format Conversion instructions**: xscvsxdp, xscvuxdp
4. **VSX Scalar Convert Double-Precision To Integer Word format with Saturate instructions**: xscvdpsdxds, xscvdpsxws, xscvdpxdpsxds, xscvdpxwxs
5. **VSX Vector Double-Precision Arithmetic instructions**: xsaddp, xssubdp, xsmdulp, xsdivdp, xssqrtsp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp, xsmaddadp, xsmaddmdp, xsmsubadp, xsmsubmdp
6. **VSX Vector Single-Precision Arithmetic instructions**: xvaddsp, xsvsubsp, xsvmulsp, xsvdivsp, xsvsqrstsp, xsvmaddsp, xsvmaddmssp, xsvmsubasp, xsvmsubmsp, xsvmaddasp, xsvmaddmssp, xsvmsubasp, xsvmsubmsp

Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 357
7.5 VSX Storage Access Operations

The VSX Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Power ISA Book I.

7.5.1 Accessing Aligned Storage Operands

The following quadword-aligned array, AH, consists of 8 halfwords.

```c
short AW[4] = { 0x0001_0203,
    0x0405_0607,
    0x0809_0A0B,
    0x0C0D_0E0F };
```

Figure 119 illustrates the big-endian storage image of array AW.

![Big-endian storage image of array AW](image1)

Figure 120 illustrates the little-endian storage image of array AW.

![Little-endian storage image of array AW](image2)

Figure 121 shows the result of loading that quadword into a VSR or, equivalently, shows the contents that must be in a VSR if storing that VSR is to produce the storage contents shown in Figure 119 for big-endian. Note that Figure shows the effect of loading the quadword from both big-endian storage and little-endian storage.

![Vector-Scalar Register contents for aligned quadword Load or Store VSX Vector](image3)
7.5.2 Accessing Unaligned Storage Operands

The following array, B, consists of 5 word elements.

```c
int B[5];
B[0] = 0x01234567;
B[1] = 0x00112233;
B[2] = 0x44556677;
B[3] = 0x8899AABB;
B[4] = 0xCCDDEEFF;
```

Figure 122 illustrates both big-endian and little-endian storage images of array B.

### Loading an Unaligned Quadword from Big-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using big-endian byte ordering.

### Loading an Unaligned Quadword from Little-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using little-endian byte ordering.

---

**# Assumptions**

- GPR[Ra] = address of B
- GPR[Rb] = 4 (index to B[1])

- lxvw4x Xt,Ra,Rb

```
Xt: 00 11 22 33 44 55 66 77 88 99 AABBCCDDEEFF
```

**Figure 123. Process to load misaligned quadword from big-endian storage using Load VSX Vector Word*4 Indexed**

---

**# Assumptions**

- GPR[A] = address of B
- GPR[B] = 4 (index to B[1])

- lxvw4x Xt,Ra,Rb

```
Xt: 00 11 22 33 44 55 66 77 88 99 AABBCCDDEEFF
```

**Figure 124. Process to load misaligned quadword from little-endian storage Load VSX Vector Word*4 Indexed**
Storing an Unaligned Quadword to Big-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using big-endian byte ordering.

### Big-endian storage image of array B

| 0x0000: | 01234567 | 00112233 | 44556677 | 8899AABB |
| 0x0010: | CC DDEE FF | 01234567 | 8899AABB | F0F1F2F3 |

### Big-endian storage image of array B

| 0x0000: | 01234567F0F1F2F3F4F5F6F7 |
| 0x0010: | FCFDFEFF |

**# Assumptions**

GPR[Ra] = address of B

GPR[Rb] = 4 \[\text{index to B}[-1]\]

```stxvw4x Xs,Ra,Rb```

**Figure 125. Process to store misaligned quadword to big-endian storage using Store VSX Vector Word*4 Indexed**

Storing an Unaligned Quadword to Little-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using little-endian byte ordering.

### Little-endian storage image of array B

| 0x0000: | 6745230133221100 |
| 0x0010: | 77665544BB AAA988 |

### Little-endian storage image of array B

| 0x0000: | 00123456789AB CDEF |
| 0x0010: | 0123456789AB CDEF |

**# Assumptions**

GPR[A] = address of B

GPR[B] = 4 \[\text{index to B}[-1]\]

```stxvw4x Xs,Ra,Rb```

**Figure 126. Process to store misaligned quadword to little-endian storage Store VSX Vector Word*4 Indexed**

### 7.5.3 Storage Access Exceptions

Storage accesses cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (Store only), or if the program attempts to access storage that is unavailable.

The address used for VSX storage accesses must be word-aligned to guarantee the access can be performed in hardware rather than result in an Alignment interrupt.
7.6 VSX Instruction Set

7.6.1 VSX Instruction Set Summary

7.6.1.1 VSX Storage Access Instructions

There are two basic forms of scalar load and scalar store instructions, word and doubleword. VSX Scalar Load instructions place a copy of the contents of the addressed word or doubleword in storage into the left-most word or doubleword element of the target VSR. The contents of the right-most element(s) of the target VSR are undefined. VSX Scalar Store instructions place a copy of the contents of the left-most word or doubleword element in the source VSR into the addressed word or doubleword in storage.

There are two basic forms of vector load and vector store instructions, a vector of 4 word elements and a vector of two doublewords. Both forms access a quadword in storage.

There is one basic form of vector load and splat instruction, doubleword. VSX Vector Load and Splat instruction places a copy of the contents of the addressed doubleword in storage into both doubleword elements of the target VSR.

7.6.1.1.1 VSX Scalar Storage Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxsdx</td>
<td>Load VSX Scalar Doubleword Indexed</td>
<td>393</td>
</tr>
<tr>
<td>lxspx</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td>394</td>
</tr>
<tr>
<td>lxsiwax</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed</td>
<td>393</td>
</tr>
<tr>
<td>lxsiwzx</td>
<td>Load VSX Scalar as Integer Word and Zero Indexed</td>
<td>394</td>
</tr>
</tbody>
</table>

Table 8. VSX Scalar Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxsdx</td>
<td>Store VSX Scalar Doubleword Indexed</td>
<td>396</td>
</tr>
<tr>
<td>stxsspx</td>
<td>Store VSX Scalar Single-Precision Indexed</td>
<td>397</td>
</tr>
<tr>
<td>stxsiwx</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td>397</td>
</tr>
</tbody>
</table>

Table 9. VSX Scalar Store Instructions

7.6.1.1.2 VSX Vector Storage Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvd2x</td>
<td>Load VSX Vector Doubleword*2 Indexed</td>
<td>395</td>
</tr>
<tr>
<td>lxvw4x</td>
<td>Load VSX Vector Word*4 Indexed</td>
<td>396</td>
</tr>
</tbody>
</table>

Table 10. VSX Vector Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvdsx</td>
<td>Load VSX Vector Doubleword and Splat Indexed</td>
<td>395</td>
</tr>
</tbody>
</table>

Table 11. VSX Vector Load and Splat Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stxvd2x</td>
<td>Store VSX Vector Doubleword*2 Indexed</td>
<td>398</td>
</tr>
<tr>
<td>stxvw4x</td>
<td>Store VSX Vector Word*4 Indexed</td>
<td>398</td>
</tr>
</tbody>
</table>

Table 12. VSX Vector Store Instructions
7.6.1.2 VSX Move Instructions

7.6.1.2.1 VSX Scalar Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsabsdp</td>
<td>VSX Scalar Absolute Value Double-Precision</td>
<td>399</td>
</tr>
<tr>
<td>xscpsgndp</td>
<td>VSX Scalar Copy Sign Double-Precision</td>
<td>411</td>
</tr>
<tr>
<td>xsnabsdp</td>
<td>VSX Scalar Negative Absolute Value Double-Precision</td>
<td>449</td>
</tr>
<tr>
<td>xsnegdp</td>
<td>VSX Scalar Negate Double-Precision</td>
<td>449</td>
</tr>
</tbody>
</table>

Table 13. VSX Scalar Double-Precision Move Instructions

7.6.1.2.2 VSX Vector Move Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvabssp</td>
<td>VSX Vector Absolute Value Single-Precision</td>
<td>481</td>
</tr>
<tr>
<td>xvcpsgnsd</td>
<td>VSX Vector Copy Sign Single-Precision</td>
<td>494</td>
</tr>
<tr>
<td>xvnabssp</td>
<td>VSX Vector Negative Absolute Value Single-Precision</td>
<td>545</td>
</tr>
<tr>
<td>xvnegsp</td>
<td>VSX Vector Negate Single-Precision</td>
<td>546</td>
</tr>
</tbody>
</table>

Table 14. VSX Vector Single-Precision Move Instructions

7.6.1.3 VSX Floating-Point Arithmetic Instructions

7.6.1.3.1 VSX Scalar Floating-Point Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsadddp</td>
<td>VSX Scalar Add Double-Precision</td>
<td>400</td>
</tr>
<tr>
<td>xsdivdp</td>
<td>VSX Scalar Divide Double-Precision</td>
<td>425</td>
</tr>
<tr>
<td>xsmuldp</td>
<td>VSX Scalar Multiply Double-Precision</td>
<td>445</td>
</tr>
<tr>
<td>xsredp</td>
<td>VSX Scalar Reciprocal Estimate Double-Precision</td>
<td>468</td>
</tr>
<tr>
<td>xsrsqtedp</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision</td>
<td>471</td>
</tr>
<tr>
<td>xssqrtdp</td>
<td>VSX Scalar Square Root Double-Precision</td>
<td>473</td>
</tr>
<tr>
<td>xssubdp</td>
<td>VSX Scalar Subtract Double-Precision</td>
<td>475</td>
</tr>
<tr>
<td>xstdivdp</td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
<td>479</td>
</tr>
<tr>
<td>xstdsqrdp</td>
<td>VSX Scalar Test for software Square Root Double-Precision</td>
<td>480</td>
</tr>
</tbody>
</table>

Table 16. VSX Scalar Double-Precision Elementary Arithmetic Instructions
### 7.6.1.3.2 VSX Vector Floating-Point Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvadddp</td>
<td>VSX Vector Add Double-Precision</td>
<td>482</td>
</tr>
<tr>
<td>xdivdp</td>
<td>VSX Vector Divide Double-Precision</td>
<td>517</td>
</tr>
<tr>
<td>xvmuldp</td>
<td>VSX Vector Multiply Double-Precision</td>
<td>541</td>
</tr>
<tr>
<td>xvredp</td>
<td>VSX Vector Reciprocal Estimate Double-Precision</td>
<td>564</td>
</tr>
<tr>
<td>xvsqrtepd</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
<td>568</td>
</tr>
<tr>
<td>xsvqrtdp</td>
<td>VSX Vector Square Root Double-Precision</td>
<td>571</td>
</tr>
<tr>
<td>xsvsubdp</td>
<td>VSX Vector Subtract Double-Precision</td>
<td>573</td>
</tr>
<tr>
<td>xvtdivdp</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
<td>577</td>
</tr>
<tr>
<td>xvttsqrtdp</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
<td>579</td>
</tr>
</tbody>
</table>

Table 20.VSX Vector Double-Precision Elementary Arithmetic Instructions
### Table 21. VSX Vector Single-Precision Elementary Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvaddsp</td>
<td>VSX Vector Add Single-Precision</td>
<td>486</td>
</tr>
<tr>
<td>xdivsp</td>
<td>VSX Vector Divide Single-Precision</td>
<td>519</td>
</tr>
<tr>
<td>xvmulspsp</td>
<td>VSX Vector Multiply Single-Precision</td>
<td>543</td>
</tr>
<tr>
<td>xvresp</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
<td>565</td>
</tr>
<tr>
<td>xvsqrtesp</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
<td>570</td>
</tr>
<tr>
<td>xvsqrtsp</td>
<td>VSX Vector Square Root Single-Precision</td>
<td>572</td>
</tr>
<tr>
<td>xvsbsp</td>
<td>VSX Vector Subtract Single-Precision</td>
<td>575</td>
</tr>
<tr>
<td>xvtsdivsp</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
<td>578</td>
</tr>
<tr>
<td>xvtsqrspsp</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
<td>579</td>
</tr>
</tbody>
</table>

### Table 22. VSX Vector Double-Precision Multiply-Add Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
<td>521</td>
</tr>
<tr>
<td>xvmaddmdp</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision</td>
<td>521</td>
</tr>
<tr>
<td>xvmsubadp</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision</td>
<td>535</td>
</tr>
<tr>
<td>xvmsubmdp</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision</td>
<td>535</td>
</tr>
<tr>
<td>xvnmmaddadp</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
<td>547</td>
</tr>
<tr>
<td>xvnmmaddmdp</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision</td>
<td>547</td>
</tr>
<tr>
<td>xvnmsubadp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision</td>
<td>555</td>
</tr>
<tr>
<td>xvnmsubmdp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision</td>
<td>555</td>
</tr>
</tbody>
</table>

### Table 23. VSX Vector Single-Precision Multiply-Add Arithmetic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvmaddaspsp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
<td>524</td>
</tr>
<tr>
<td>xvmaddmsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
<td>524</td>
</tr>
<tr>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
<td>538</td>
</tr>
<tr>
<td>xvmsubmsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
<td>538</td>
</tr>
<tr>
<td>xvnmmaddasp</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision</td>
<td>552</td>
</tr>
<tr>
<td>xvnmmaddmsp</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision</td>
<td>552</td>
</tr>
<tr>
<td>xvnmsubasp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
<td>558</td>
</tr>
<tr>
<td>xvnmsubmsp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision</td>
<td>558</td>
</tr>
</tbody>
</table>
7.6.1.4 VSX Floating-Point Compare Instructions

7.6.1.4.1 VSX Scalar Floating-Point Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscmpodp</td>
<td>VSX Scalar Compare Ordered Double-Precision</td>
<td>407</td>
</tr>
<tr>
<td>xscmpudp</td>
<td>VSX Scalar Compare Unordered Double-Precision</td>
<td>409</td>
</tr>
</tbody>
</table>

Table 24. VSX Scalar Compare Double-Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsmaxdp</td>
<td>VSX Scalar Maximum Double-Precision</td>
<td>435</td>
</tr>
<tr>
<td>xsmindp</td>
<td>VSX Scalar Minimum Double-Precision</td>
<td>437</td>
</tr>
</tbody>
</table>

Table 25. VSX Scalar Double-Precision Maximum/Minimum Instructions

7.6.1.4.2 VSX Vector Floating-Point Compare Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcmpedp</td>
<td>VSX Vector Compare Equal To Double-Precision</td>
<td>488</td>
</tr>
<tr>
<td>xvcmpgdtp</td>
<td>VSX Vector Compare Greater Than or Equal To Double-Precision</td>
<td>490</td>
</tr>
<tr>
<td>xvcmpgtdp</td>
<td>VSX Vector Compare Greater Than Double-Precision</td>
<td>492</td>
</tr>
</tbody>
</table>

Table 26. VSX Vector Compare Double-Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcmpesp</td>
<td>VSX Vector Compare Equal To Single-Precision</td>
<td>489</td>
</tr>
<tr>
<td>xvcmpgesp</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision</td>
<td>491</td>
</tr>
<tr>
<td>xvcmpgtsp</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
<td>493</td>
</tr>
</tbody>
</table>

Table 27. VSX Vector Compare Single-Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvmaxdp</td>
<td>VSX Vector Maximum Double-Precision</td>
<td>527</td>
</tr>
<tr>
<td>xvmindp</td>
<td>VSX Vector Minimum Double-Precision</td>
<td>531</td>
</tr>
</tbody>
</table>

Table 28. VSX Vector Double-Precision Maximum/Minimum Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvmaxsp</td>
<td>VSX Vector Maximum Single-Precision</td>
<td>529</td>
</tr>
<tr>
<td>xvmbsp</td>
<td>VSX Vector Minimum Single-Precision</td>
<td>533</td>
</tr>
</tbody>
</table>

Table 29. VSX Vector Single-Precision Maximum/Minimum Instructions
7.6.1.5 VSX DP-SP Conversion Instructions

7.6.1.5.1 VSX Scalar DP-SP Conversion Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvdpsp</td>
<td>VSX Scalar round and Convert Double-Precision to Single-Precision format</td>
<td>412</td>
</tr>
<tr>
<td>xscvsdp</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision format</td>
<td>422</td>
</tr>
</tbody>
</table>

Table 30. VSX Scalar DP-SP Conversion Instructions

7.6.1.5.2 VSX Vector DP-SP Conversion Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvdpsp</td>
<td>VSX Vector round and Convert Double-Precision to Single-Precision format</td>
<td>495</td>
</tr>
<tr>
<td>xvcvsdp</td>
<td>VSX Vector Convert Single-Precision to Double-Precision format</td>
<td>504</td>
</tr>
</tbody>
</table>

Table 31. VSX Vector DP-SP Conversion Instructions

7.6.1.6 VSX Integer Conversion Instructions

7.6.1.6.1 VSX Scalar Integer Conversion Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvdpsdxds</td>
<td>VSX Scalar truncate Double-Precision to integer and Convert to Signed Fixed-Point Doubleword format with Saturate</td>
<td>413</td>
</tr>
<tr>
<td>xscvdpsxws</td>
<td>VSX Scalar truncate Double-Precision to integer and Convert to Signed Fixed-Point Word format with Saturate</td>
<td>416</td>
</tr>
<tr>
<td>xscvdpuxds</td>
<td>VSX Scalar truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Doubleword format with Saturate</td>
<td>418</td>
</tr>
<tr>
<td>xscvdpuxws</td>
<td>VSX Scalar truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Word format with Saturate</td>
<td>420</td>
</tr>
</tbody>
</table>

Table 32. VSX Scalar Convert Double-Precision to Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvsxdsp</td>
<td>VSX Scalar Convert Signed Fixed-Point Doubleword to floating-point format and round to Double-Precision</td>
<td>423</td>
</tr>
<tr>
<td>xscvuxdsp</td>
<td>VSX Scalar Convert Unsigned Fixed-Point Doubleword to floating-point format and round to Single-Precision</td>
<td>424</td>
</tr>
</tbody>
</table>

Table 33. VSX Scalar Convert Integer to Double-Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscvsxdsp</td>
<td>VSX Scalar Convert Signed Fixed-Point Doubleword to floating-point format and round to Single-Precision</td>
<td>423</td>
</tr>
<tr>
<td>xscvuxdsp</td>
<td>VSX Scalar Convert Unsigned Fixed-Point Doubleword to floating-point format and round to Single-Precision</td>
<td>424</td>
</tr>
</tbody>
</table>

Table 34. VSX Scalar Convert Integer to Single-Precision Instructions
7.6.1.6.2 VSX Vector Integer Conversion Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvdpsxds</td>
<td>VSX Vector truncate Double-Precision to integer and Convert to Signed Fixed-Point</td>
<td>496</td>
</tr>
<tr>
<td></td>
<td>Doubleword format with Saturate</td>
<td></td>
</tr>
<tr>
<td>xvcvdpsxws</td>
<td>VSX Vector truncate Double-Precision to integer and Convert to Signed Fixed-Point Word format with Saturate</td>
<td>498</td>
</tr>
<tr>
<td>xvcvdpuxds</td>
<td>VSX Vector truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Doubleword format with Saturate</td>
<td>500</td>
</tr>
<tr>
<td>xvcvdpuxws</td>
<td>VSX Vector truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Word format with Saturate</td>
<td>502</td>
</tr>
</tbody>
</table>

Table 35. VSX Vector Convert Double-Precision to Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvpsxds</td>
<td>VSX Vector truncate Single-Precision to integer and Convert to Signed Fixed-Point Doubleword format with Saturate</td>
<td>505</td>
</tr>
<tr>
<td>xvcvpsxws</td>
<td>VSX Vector truncate Single-Precision to integer and Convert to Signed Fixed-Point Word format with Saturate</td>
<td>507</td>
</tr>
<tr>
<td>xvcvpsuxds</td>
<td>VSX Vector truncate Single-Precision to integer and Convert to Unsigned Fixed-Point Doubleword format with Saturate</td>
<td>509</td>
</tr>
<tr>
<td>xvcvpsuxws</td>
<td>VSX Vector truncate Single-Precision to integer and Convert to Unsigned Fixed-Point Word format with Saturate</td>
<td>511</td>
</tr>
</tbody>
</table>

Table 36. VSX Vector Convert Single-Precision to Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvsxddp</td>
<td>VSX Vector Convert and round Signed Fixed-Point Doubleword to Double-Precision format</td>
<td>513</td>
</tr>
<tr>
<td>xvcvxxdp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Double-Precision format</td>
<td>514</td>
</tr>
<tr>
<td>xvcvuxddp</td>
<td>VSX Vector Convert and round Unsigned Fixed-Point Doubleword to Double-Precision format</td>
<td>515</td>
</tr>
<tr>
<td>xvcvuxwdp</td>
<td>VSX Vector Convert Unsigned Fixed-Point Word to Double-Precision format</td>
<td>516</td>
</tr>
</tbody>
</table>

Table 37. VSX Vector Convert Integer to Double-Precision Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvcvxsxdp</td>
<td>VSX Vector Convert and round Signed Fixed-Point Doubleword to Single-Precision format</td>
<td>513</td>
</tr>
<tr>
<td>xvcvxswsp</td>
<td>VSX Vector Convert and round Signed Fixed-Point Word to Single-Precision format</td>
<td>514</td>
</tr>
<tr>
<td>xvcvuxxdp</td>
<td>VSX Vector Convert and round Unsigned Fixed-Point Doubleword to Single-Precision format</td>
<td>515</td>
</tr>
<tr>
<td>xvcvuxwsp</td>
<td>VSX Vector Convert and round Unsigned Fixed-Point Word to Single-Precision format</td>
<td>516</td>
</tr>
</tbody>
</table>

Table 38. VSX Vector Convert Integer to Single-Precision Instructions
### 7.6.1.7 VSX Round to Floating-Point Integer Instructions

#### 7.6.1.7.1 VSX Scalar Round to Floating-Point Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsrdpi</td>
<td>VSX Scalar Round to Double-Precision Integer using round to Nearest Away</td>
<td>464</td>
</tr>
<tr>
<td>xsrdpic</td>
<td>VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode</td>
<td>465</td>
</tr>
<tr>
<td>xsrdpim</td>
<td>VSX Scalar Round to Double-Precision Integer using round towards -Infinity rounding mode</td>
<td>466</td>
</tr>
<tr>
<td>xsrdpiz</td>
<td>VSX Scalar Round to Double-Precision Integer using round towards Zero rounding mode</td>
<td>467</td>
</tr>
</tbody>
</table>

Table 39. VSX Scalar Round to Double-Precision Integer Instructions

#### 7.6.1.7.2 VSX Vector Round to Floating-Point Integer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvrdpi</td>
<td>VSX Vector Round to Double-Precision Integer using round to Nearest Away</td>
<td>561</td>
</tr>
<tr>
<td>xvrdpic</td>
<td>VSX Vector Round to Double-Precision Integer Exact using Current rounding mode</td>
<td>561</td>
</tr>
<tr>
<td>xvrdpim</td>
<td>VSX Vector Round to Double-Precision Integer using round towards -Infinity rounding mode</td>
<td>562</td>
</tr>
<tr>
<td>xvrdpiz</td>
<td>VSX Vector Round to Double-Precision Integer using round towards Zero rounding mode</td>
<td>563</td>
</tr>
</tbody>
</table>

Table 40. VSX Vector Round to Double-Precision Integer Instructions

#### 7.6.1.8 VSX Logical Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxland</td>
<td>VSX Logical AND</td>
<td>580</td>
</tr>
<tr>
<td>xxlandc</td>
<td>VSX Logical AND with Complement</td>
<td>580</td>
</tr>
<tr>
<td>xxlnor</td>
<td>VSX Logical NOR</td>
<td>582</td>
</tr>
<tr>
<td>xxlor</td>
<td>VSX Logical OR</td>
<td>583</td>
</tr>
<tr>
<td>xxlxor</td>
<td>VSX Logical XOR</td>
<td>583</td>
</tr>
</tbody>
</table>

Table 42. VSX Logical Instructions

#### 7.6.1.8.1 VSX Vector Select Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxsel</td>
<td>VSX Select</td>
<td>585</td>
</tr>
</tbody>
</table>

Table 43. VSX Vector Select Instruction
### 7.6.1.9 VSX Permute Instructions

Table 44. VSX Merge Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxmrghw</td>
<td>VSX Merge High Word</td>
<td>584</td>
</tr>
<tr>
<td>xxmrglw</td>
<td>VSX Merge Low Word</td>
<td>584</td>
</tr>
</tbody>
</table>

Table 45. VSX Splat Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxspltw</td>
<td>VSX Splat Word</td>
<td>586</td>
</tr>
</tbody>
</table>

Table 46. VSX Permute Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxpermdi</td>
<td>VSX Permute Doubleword Immediate</td>
<td>585</td>
</tr>
</tbody>
</table>

Table 47. VSX Shift Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxslwdi</td>
<td>VSX Shift Left Double by Word Immediate</td>
<td>586</td>
</tr>
</tbody>
</table>
7.6.2 VSX Instruction Description Conventions

7.6.2.1 VSX Instruction RTL Operators

x.bit[y]
    Return the contents of bit y of x.

x.bit[y:z]
    Return the contents of bits y:z of x.

x.word[y]
    Return the contents of word element y of x.

x.word[y:z]
    Return the contents of word elements y:z of x.

x.dword[y]
    Return the contents of doubleword element y of x.

x.dword[y:z]
    Return the contents of doubleword elements y:z of x.

x = y
    The value of y is placed into x.

x |= y
    The value of y is ORed with the value x and placed into x.

~x
    Return the one’s complement of x.

!x
    Return 1 if the contents of x are equal to 0, otherwise return 0.

x | | y
    Return the value of x concatenated with the value of y. For example, 0b010 || 0b111 is the same as 0b010111.

x ^ y
    Return the value of x exclusive ORed with the value of y.

x ? y : z
    If the value of x is true, return the value of y, otherwise return the value z.

x+y
    x and y are integer values.
    Return the sum of x and y.

x–y
    x and y are integer values.
    Return the difference of x and y.
7.6.2.2 VSX Instruction RTL Function Calls

AddDP(x, y)

x and y are double-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an Infinity and y is an Infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x and y, having unbounded range and precision.

AddSP(x, y)

x and y are single-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an Infinity and y is an Infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x added to y, having unbounded range and precision.

ClassDP(x, y)

Return a 5-bit characterization of the double-precision floating-point number x.

0b10001 = Quiet NaN
0b01001 = -Infinity
0b01000 = -Normalized Number
0b11000 = -Denormalized Number
0b10010 = -Zero
0b00010 = +Zero
0b10100 = +Denormalized Number
0b00100 = +Normalized Number
0b00101 = +Infinity

ClassSP(x, y)

Return a 5-bit characterization of the single-precision floating-point number x.

0b10001 = Quiet NaN
0b01001 = -Infinity
0b01000 = -Normalized Number
0b11000 = -Denormalized Number
0b10010 = -Zero
0b00010 = +Zero
0b10100 = +Denormalized Number
0b00100 = +Normalized Number
0b00101 = +Infinity
CompareEQDP(x, y)
  x and y are double-precision floating-point values.

  If x or y is a NaN, return 0.
  Otherwise, if x is equal to y, return 1.
  Otherwise, return 0.

CompareEQSP(x, y)
  x and y are single-precision floating-point values.

  If x or y is a NaN, return 0.
  Otherwise, if x is equal to y, return 1.
  Otherwise, return 0.

CompareGTDP(x, y)
  x and y are double-precision floating-point values.

  If x or y is a NaN, return 0.
  Otherwise, if x is greater than y, return 1.
  Otherwise, return 0.

CompareGTSP(x, y)
  x and y are single-precision floating-point values.

  If x or y is a NaN, return 0.
  Otherwise, if x is greater than y, return 1.
  Otherwise, return 0.

CompareLTDP(x, y)
  x and y are double-precision floating-point values.

  If x or y is a NaN, return 0.
  Otherwise, if x is less than y, return 1.
  Otherwise, return 0.

CompareLTSP(x, y)
  x and y are single-precision floating-point values.

  If x or y is a NaN, return 0.
  Otherwise, if x is less than y, return 1.
  Otherwise, return 0.

ConvertDPtoSD(x)
  x is a double-precision integer value.

  If x is a NaN,
    return 0x8000_0000_0000_0000, vxcvi_flag is set to 1, and vxsnan_flag is set to 1 if x is an SNaN.
  Otherwise, do the following.
    If x is greater than 2^{63}-1,
      return 0x7FFF_FFFF_FFFF_FFFF and vxcvi_flag is set to 1.
    Otherwise, if x is less than -2^{63},
      return 0x8000_0000_0000_0000 and vxcvi_flag is set to 1.
  Otherwise, return the value x in 64-bit signed integer format.
ConvertDPtoSP(x)

x is a double-precision floating-point value.

If x is an SNaN, \texttt{vxsnan\_flag} is set to 1.

If x is a SNaN, returns x, converted to a QNaN, in single-precision floating-point format.

Otherwise, if x is a QNaN, an Infinity, or a Zero, returns x in single-precision floating-point format.

Otherwise, returns x, rounded to single-precision using the rounding mode specified in RN, in single-precision floating-point format.

\texttt{ox\_flag} is set to 1 if rounding x resulted in an Overflow exception.
\texttt{ux\_flag} is set to 1 if rounding x resulted in an Underflow exception.
\texttt{xx\_flag} is set to 1 if rounding x returns an inexact result.
\texttt{inc\_flag} is set to 1 if the significand of the result was incremented during rounding.

ConvertDPtoSP\_NS(x)

x is a single-precision floating-point value represented in double-precision format.

Returns x in single-precision format.

\begin{verbatim}
sign ← x.bit[0]
exponent ← x.bit[1:11]
fraction ← 0b1 || x.bit[12:63] // implicit bit set to 1 (for now)

if (exponent == 0) & (fraction.bit[1:52] != 0) then do // DP Denormal operand
    exponent ← 0b000_0000_0001 // exponent override to DP Emin = 1
    fraction.bit[0] ← 0b0 // implicit bit override to 0
end

if (exponent < 897) && (fraction != 0) then do // SP tiny operand
    fraction ← fraction >>ui (897 - exponent) // denormalize until exponent = SP Emin
    exponent ← 0b011_1000_0000 // exponent override to SP Emin-1 = 896
end

return(sign || exponent.bit[0] || exponent.bit[4:10] || fraction.bit[1:23])
\end{verbatim}

Programming Note

If x is not representable in single-precision, some exponent and/or significand bits will be discarded, likely producing undesirable results. The low-order 29 bits of the significand of x are discarded, more if the unbiased exponent of x is less than -126 (i.e., denormal). Finite values of x having an unbiased exponent less than -155 will return a result of Zero. Finite values of x having an unbiased exponent greater than +127 will result in discarding significant bits of the exponent. SNaN inputs having no significant bits in the upper 23 bits of the significand will return Infinity as the result. No status is set for any of these cases.
ConvertDPtoSW(x)
   x is a double-precision integer value.

   If x is a NaN,
   return 0x8000_0000,
   vxcvi_flag is set to 1, and
   vxsnan_flag is set to 1 if x is an SNaN

   Otherwise, do the following.
   If x is greater than $2^{31}-1$,
   return 0x7FFF_FFFF and
   vxcvi_flag is set to 1.

   Otherwise, if x is less than $-2^{31}$,
   return 0x8000_0000 and
   vxcvi_flag is set to 1.

   Otherwise, return the value x in 32-bit signed integer format.

ConvertDPtoUD(x)
   x is a double-precision integer value.

   If x is a NaN,
   return 0x0000_0000_0000_0000,
   vxcvi_flag is set to 1, and
   vxsnan_flag is set to 1 if x is an SNaN

   Otherwise, do the following.
   If x is greater than $2^{64}-1$,
   return 0xFFFF_FFFF_FFFF_FFFF and
   vxcvi_flag is set to 1.

   Otherwise, if x is less than 0,
   return 0x0000_0000_0000_0000 and
   vxcvi_flag is set to 1.

   Otherwise, return the value x in 64-bit unsigned integer format.

ConvertDPtoUW(x)
   x is a double-precision integer value.

   If x is a NaN,
   return 0x0000_0000,
   vxcvi_flag is set to 1, and
   vxsnan_flag is set to 1 if x is an SNaN

   Otherwise, do the following.
   If x is greater than $2^{32}-1$,
   return 0xFFFF_FFFF and
   vxcvi_flag is set to 1.

   Otherwise, if x is less than 0,
   return 0x0000_0000 and
   vxcvi_flag is set to 1.

   Otherwise, return the value x in 32-bit unsigned integer format.

ConvertFPtoDP(x)
   Return the floating-point value x in DP format.
ConvertFPtoSP(x)
Return the floating-point value x in single-precision format.

ConvertSDtoFP(x)
x is a 64-bit signed integer value.
Return the value x converted to floating-point format having unbounded significand precision.

ConvertSPtoDP_NS(x)
x is a single-precision floating-point value.
Returns x in double-precision format.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exponent} & \leftarrow (x.\text{bit}[1] \lor \neg x.\text{bit}[1] \lor \neg x.\text{bit}[1] \lor \neg x.\text{bit}[1] \lor x.\text{bit}[2:8]) \\
\text{fraction} & \leftarrow 0b0 \lor x.\text{bit}[9:31] \lor 0b0_0000_0000_0000_0000_0000_0000_0000
\end{align*}
\]

if (x.bit[1:8] == 255) then do
   // Infinity or NaN operand
   exponent \leftarrow 2047
   // override exponent to DP Emax+1
end

else if (x.bit[1:8] == 0) \&\& (fraction == 0) then do
   // SP Zero operand
   exponent \leftarrow 0
   // override exponent to DP Emin-1
end

else if (x.bit[1:8] == 0) \&\& (fraction != 0) then do
   // SP Denormal operand
   exponent \leftarrow 897
   // override exponent to SP Emin
   do while (fraction.bit[0] == 0)
      // normalize operand
      fraction \leftarrow fraction << 1
      exponent \leftarrow exponent + 1
   end
end

return(sign \lor exponent \lor fraction.bit[1:52])
ConvertSP64toSP(x)

x is a single-precision floating-point value in double-precision format.

Returns the value x in single-precision format. x must be representable in single-precision, or else result returned is undefined. x may require denormalization. No rounding is performed. If x is a SNaN, it is converted to a single-precision SNaN having the same payload as x.

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exp} & \leftarrow x.\text{bit}[1:11] - 1023 \\
\text{frac} & \leftarrow x.\text{bit}[12:63]
\end{align*}
\]

\[
\begin{align*}
\text{if} & \quad (\text{exp} = -1023) \land (\text{frac} = 0) \land (\text{sign} = 0) \quad \text{then return}(0x0000_0000) \quad \text{// +Zero} \\
\text{else if} & \quad (\text{exp} = -1023) \land (\text{frac} = 0) \land (\text{sign} = 1) \quad \text{then return}(0x8000_0000) \quad \text{// -Zero} \\
\text{else if} & \quad (\text{exp} = -1023) \land (\text{frac} \neq 0) \quad \text{then return}(0xUUUU_UUUU) \quad \text{// DP denorm} \\
\text{else if} & \quad (\text{exp} < -126) \quad \text{then do} \quad \text{// denormalization required} \\
& \quad \text{msb} = 1 \\
& \quad \text{do while} \quad (\text{exp} < -126) \quad \text{// denormalize operand until exp=Emint} \\
& \quad \quad \text{frac.}\text{bit}[1:51] \leftarrow \text{frac.}\text{bit}[0:50] \\
& \quad \quad \text{frac.}\text{bit}[0] \leftarrow \text{msb} \\
& \quad \quad \text{msb} \leftarrow 0 \\
& \quad \quad \text{exp} \leftarrow \text{exp} + 1 \\
& \quad \text{end} \\
& \quad \text{if} \quad (\text{frac} = 0) \quad \text{then return}(0xUUUU_UUUU) \quad \text{// value not representable in SP format} \\
& \quad \text{else do} \quad \text{// return denormal SP} \\
& \quad & \quad \text{result.}\text{bit}[0] \leftarrow \text{sign} \\
& \quad & \quad \text{result.}\text{bit}[1:8] \leftarrow 0 \\
& \quad & \quad \text{result.}\text{bit}[9:31] \leftarrow \text{frac.}\text{bit}[0:22] \\
& \quad \quad \text{return(result)} \\
& \quad \text{end} \\
\text{else if} & \quad (\text{exp} = +1024) \land (\text{frac} = 0) \land (\text{sign} = 0) \quad \text{then return}(0x7F80_0000) \quad \text{// +Infinity} \\
\text{else if} & \quad (\text{exp} = +1024) \land (\text{frac} = 0) \land (\text{sign} = 1) \quad \text{then return}(0xFF80_0000) \quad \text{// -Infinity} \\
\text{else if} & \quad (\text{exp} = +1024) \land (\text{frac} \neq 0) \quad \text{then do} \quad \text{// QNaN or SNaN} \\
& \quad & \quad \text{result.}\text{bit}[0] \leftarrow \text{sign} \\
& \quad & \quad \text{result.}\text{bit}[1:8] \leftarrow 255 \\
& \quad & \quad \text{result.}\text{bit}[9:31] \leftarrow \text{frac.}\text{bit}[0:22] \\
& \quad \quad \text{return(result)} \\
& \quad \text{end} \\
\text{else if} & \quad (\text{exp} < +1024) \land (\text{exp} > +126) \quad \text{then return}(0xUUUU_UUUU) \quad \text{// overflow} \\
\text{else do} \quad \text{// normal value} \\
& \quad \quad \text{result.}\text{bit}[0] \leftarrow \text{sign} \\
& \quad \quad \text{result.}\text{bit}[1:8] \leftarrow \text{exp.}\text{bit}[4:11] + 127 \\
& \quad \quad \text{result.}\text{bit}[9:31] \leftarrow \text{frac.}\text{bit}[0:22] \\
& \quad \quad \text{return(result)} \\
& \quad \text{end}
\end{align*}
\]

ConvertSPtoDP(x)

x is a single-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.

If x is an SNaN, return x represented as a QNaN in double-precision floating-point format. Otherwise, if x is a QNaN, return x in double-precision floating-point format. Otherwise, the value x in double-precision floating-point format.

ConvertSPtoSD(x)

x is a single-precision integer value.

If x is a NaN,

\[
\text{return } 0x8000_0000_0000_0000 \quad \text{and} \\
\text{vxvcvi_flag is set to 1, and} \\
\text{vxsnan_flag is set to 1 if x is an SNaN}
\]

Otherwise, do the following.
If \( x \) is greater than \( 2^{63}-1 \),
\[
\text{return } 0x7FFF_FFFF_FFFF_FFFF \text{ and } \text{vxcvi\_flag is set to 1.}
\]

Otherwise, if \( x \) is less than \(-2^{63} \),
\[
\text{return } 0x8000_0000_0000_0000 \text{ and } \text{vxcvi\_flag is set to 1.}
\]

Otherwise, return the value \( x \) in 64-bit signed integer format.

\textbf{ConvertSPtoSP64}(x)

\( x \) is a floating-point value in single-precision format.

Returns the value \( x \) in double-precision format. If \( x \) is a SNaN, it is converted to a double-precision SNaN having the same payload as \( x \).

\[
\begin{align*}
\text{sign} & \leftarrow x.\text{bit}[0] \\
\text{exp} & \leftarrow x.\text{bit}[1:8] - 127 \\
\text{frac} & \leftarrow x.\text{bit}[9:31]
\end{align*}
\]

\[
\begin{align*}
\text{if (exp = –127) \&\&(frac != 0) then do} & // \text{Normalize the Denormal value} \\
\text{msb} & \leftarrow frac.\text{bit}[0] \\
\text{frac} & \leftarrow frac \ll 1 \\
\text{do while (msb = 0)} & \\
\text{msb} & \leftarrow frac.\text{bit}[0] \\
\text{frac} & \leftarrow frac \ll 1 \\
\text{exp} & \leftarrow exp - 1 \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{else if (exp = –127) \&\&(frac = 0) then exp} & \leftarrow –1023 // \text{Zero value} \\
\text{else if (exp = +128) \&\& then exp} & \leftarrow +1024 // \text{Infinity, NaN}
\end{align*}
\]

\[
\begin{align*}
\text{result.}\text{bit}[0] & \leftarrow \text{sign} \\
\text{result.}\text{bit}[1:11] & \leftarrow \text{exp} + 1023 \\
\text{result.}\text{bit}[12:34] & \leftarrow \text{frac} \\
\text{result.}\text{bit}[35:63] & \leftarrow 0 \\
\text{return(result)}
\end{align*}
\]

\textbf{ConvertSPtoSW}(x)

\( x \) is a single-precision integer value.

If \( x \) is a NaN,
\[
\text{return } 0x8000_0000, \text{vxcvi\_flag is set to 1, and} \\
\text{vxsnan\_flag is set to 1 if } x \text{ is an SNaN}
\]

Otherwise, do the following.

If \( x \) is greater than \( 2^{31}-1 \),
\[
\text{return } 0x7FFF_FFFF \text{ and } \text{vxcvi\_flag is set to 1.}
\]

Otherwise, if \( x \) is less than \(-2^{31} \),
\[
\text{return } 0x8000_0000 \text{ and } \text{vxcvi\_flag is set to 1.}
\]

Otherwise, return the value \( x \) in 32-bit signed integer format.

\textbf{ConvertSPtoUD}(x)

\( x \) is a single-precision integer value.

If \( x \) is a NaN,
\[
\text{return } 0x0000_0000_0000_0000, \\
\text{vxcvi\_flag is set to 1.}
\]

Otherwise, do the following.

If \( x \) is greater than \( 2^{31}-1 \),
\[
\text{return } 0x7FFF_FFFF \text{ and } \text{vxcvi\_flag is set to 1.}
\]

Otherwise, if \( x \) is less than \(-2^{31} \),
\[
\text{return } 0x8000_0000 \text{ and } \text{vxcvi\_flag is set to 1.}
\]

Otherwise, return the value \( x \) in 32-bit signed integer format.
vxci_flag is set to 1, and
vxsnan_flag is set to 1 if x is an SNaN

Otherwise, do the following.
If x is greater than $2^{54}-1$,
    return 0xFFFF_FFFF_FFFF_FFFF and
    vxci_flag is set to 1.

Otherwise, if x is less than 0,
    return 0x0000_0000_0000_0000 and
    vxci_flag is set to 1.

Otherwise, return the value x in 64-bit unsigned integer format.

ConvertSPtoUW(x)
  x is a single-precision integer value.

  If x is a NaN,
    return 0x0000_0000,
    vxci_flag is set to 1, and
    vxsnan_flag is set to 1 if x is an SNaN

  Otherwise, do the following.
    If x is greater than $2^{32}-1$,
        return 0xFFFF_FFFF and
        vxci_flag is set to 1.

    Otherwise, if x is less than 0,
        return 0x0000_0000 and
        vxci_flag is set to 1.

  Otherwise, return the value x in 32-bit unsigned integer format.

ConvertSWtoFP(x)
  x is a 32-bit signed integer value.

  Return the value x converted to floating-point format having unbounded significand precision.

ConvertUdtoFP(x)
  x is a 64-bit unsigned integer value.

  Return the value x converted to floating-point format having unbounded significand precision.

ConvertUWtoFP(x)
  x is a 32-bit unsigned integer value.

  Return the value x converted to floating-point format having unbounded significand precision.
**DivideDP(x,y)**

x and y are double-precision floating-point values.

- If x or y is an SNaN, `vxsnan_flag` is set to 1.
- If x is a Zero and y is a Zero, `vxzdz_flag` is set to 1.
- If x is a finite, nonzero value and y is a Zero, `zx_flag` is set to 1.
- If x is an Infinity and y is an Infinity, `vxidi_flag` is set to 1.
- If x is a QNaN, return x.
- Otherwise, if x is an SNaN, return x represented as a QNaN.
- Otherwise, if y is a QNaN, return y.
- Otherwise, if y is an SNaN, return y represented as a QNaN.
- Otherwise, if x is a Zero and y is a Zero, return the standard QNaN.
- Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
- Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
- Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
- Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

**DivideSP(x,y)**

x and y are single-precision floating-point values.

- If x or y is an SNaN, `vxsnan_flag` is set to 1.
- If x is a Zero and y is a Zero, `vxzdz_flag` is set to 1.
- If x is a finite, nonzero value and y is a Zero, `zx_flag` is set to 1.
- If x is an Infinity and y is an Infinity, `vxidi_flag` is set to 1.
- If x is a QNaN, return x.
- Otherwise, if x is an SNaN, return x represented as a QNaN.
- Otherwise, if y is a QNaN, return y.
- Otherwise, if y is an SNaN, return y represented as a QNaN.
- Otherwise, if x is a Zero and y is a Zero, return the standard QNaN.
- Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
- Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
- Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
- Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

**DenormDP(x)**

x is a floating-point value having unbounded range and precision.

Return the value x with its significand shifted right by a number of bits equal to the difference of the -1022 and the unbiased exponent of x, and its unbiased exponent set to -1022.

**DenormSP(x)**

x is a floating-point value having unbounded range and precision.

Return the value x with its significand shifted right by a number of bits equal to the difference of the -126 and the unbiased exponent of x, and its unbiased exponent set to -126.

**IsInf(x)**

Return 1 if x is an Infinity, otherwise return 0.

**isNaN(x)**

Return 1 if x is either an SNaN or a QNaN, otherwise return 0.
IsNeg(x)
Return 1 if x is a negative, nonzero value, otherwise return 0.

IsSNaN(x)
Return 1 if x is an SNaN, otherwise return 0.

IsZero(x)
Return 1 if x is a Zero, otherwise return 0.

MaximumDP(x,y)
x and y are double-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than –0.

MaximumSP(x,y)
x and y are single-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than –0.

MEM(x,y)
Contents of a sequence of y bytes of storage. The sequence depends on the endianness of the storage access as follows.
– For big-endian storage accesses, the sequence starts with the byte at address x and ends with the byte at address x+y-1.
– For little-endian storage accesses, the sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MinimumDP(x,y)
x and y are double-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where –0 is considered less than +0.
MinimumSP(x,y)
   x and y are single-precision floating-point values.

   If x or y is an SNaN, vxsnan_flag is set to 1.

   If x is a QNaN and y is not a NaN, return y.
   Otherwise, if x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if y is a QNaN, return x.
   Otherwise, if y is an SNaN, return y represented as a QNaN.
   Otherwise, return the lesser of x and y, where –0 is considered less than +0.

MultiplyAddDP(x,y,z)
   x, y and z are double-precision floating-point values.

   If x, y or z is an SNaN, vxsnan_flag is set to 1.

   If x is a Zero and y, is an Infinity or x is an Infinity and y is an Zero, vximz_flag is set to 1.

   If the product of x and y is an Infinity and z is an Infinity of the opposite sign, vxisi_flag is set to 1.

   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if z is a QNaN, return z.
   Otherwise, if z is an SNaN, return z represented as a QNaN.
   Otherwise, if y is a QNaN, return y.
   Otherwise, if y is an SNaN, return y represented as a QNaN.
   Otherwise, if y is a QNaN, return the standard QNaN.
   Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
   Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyAddSP(x,y,z)
   x, y and z are single-precision floating-point values.

   If x, y or z is an SNaN, vxsnan_flag is set to 1.

   If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, vximz_flag is set to 1.

   If the product of x and y is an Infinity and z is an Infinity of the opposite sign, vxisi_flag is set to 1.

   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if z is a QNaN, return z.
   Otherwise, if z is an SNaN, return z represented as a QNaN.
   Otherwise, if y is a QNaN, return y.
   Otherwise, if y is an SNaN, return y represented as a QNaN.
   Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
   Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
   Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.
MultiplyDP(x, y)
  x and y are double-precision floating-point values.

  If x or y is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, vximz_flag is set to 1.

  If x is a QNaN, return x.
  Otherwise, if x is an SNaN, return x represented as a QNaN.
  Otherwise, if y is a QNaN, return y.
  Otherwise, if y is an SNaN, return y represented as a QNaN.
  Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
  Otherwise, return the normalized product of x and y, having unbounded range and precision.

MultiplySP(x, y)
  x and y are single-precision floating-point values.

  If x or y is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, vximz_flag is set to 1.

  If x is a QNaN, return x.
  Otherwise, if x is an SNaN, return x represented as a QNaN.
  Otherwise, if y is a QNaN, return y.
  Otherwise, if y is an SNaN, return y represented as a QNaN.
  Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
  Otherwise, return the normalized product of x and y, having unbounded range and precision.

NegateDP(x)
  If the double-precision floating-point value x is a NaN, return x.
  Otherwise, return the double-precision floating-point value x with its sign bit complemented.

NegateSP(x)
  If the single-precision floating-point value x is a NaN, return x.
  Otherwise, return the single-precision floating-point value x with its sign bit complemented.

ReciprocalEstimateDP(x)
  x is a double-precision floating-point value.

  If x is an SNaN, vxsnan_flag is set to 1.

  If x is a Zero, zx_flag is set to 1.

  If x is a QNaN, return x.
  Otherwise, if x is an SNaN, return x represented as a QNaN.
  Otherwise, if x is a Zero, return an Infinity with the sign of x.
  Otherwise, if x is an Infinity, return a Zero with the sign of x.
  Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.
ReciprocalEstimateSP(x)
   x is a single-precision floating-point value.
   
   If x is an SNaN, \texttt{vxsnan\_flag} is set to 1.
   
   If x is a Zero, \texttt{zx\_flag} is set to 1.
   
   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if x is a Zero, return an Infinity with the sign of x.
   Otherwise, if x is an Infinity, return a Zero with the sign of x.
   Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.

ReciprocalSquareRootEstimateDP(x)
   x is a double-precision floating-point value.
   
   If x is an SNaN, \texttt{vxsnan\_flag} is set to 1.
   
   If x is a Zero, \texttt{zx\_flag} is set to 1.
   
   If x is a negative, nonzero number, \texttt{vxsqrt\_flag} is set to 1.
   
   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if x is a negative, nonzero value, return the default QNaN.
   Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

ReciprocalSquareRootEstimateSP(x)
   x is a single-precision floating-point value.
   
   If x is an SNaN, \texttt{vxsnan\_flag} is set to 1.
   
   If x is a Zero, \texttt{zx\_flag} is set to 1.
   
   If x is a negative, nonzero number, \texttt{vxsqrt\_flag} is set to 1.
   
   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if x is a negative, nonzero value, return the default QNaN.
   Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

reset\_xflags()
   \texttt{vxsnan\_flag} is set to 0.
   \texttt{vximz\_flag} is set to 0.
   \texttt{vxidi\_flag} is set to 0.
   \texttt{vxisi\_flag} is set to 0.
   \texttt{vxzdz\_flag} is set to 0.
   \texttt{vxsqrt\_flag} is set to 0.
   \texttt{vxcvi\_flag} is set to 0.
   \texttt{vxvc\_flag} is set to 0.
   \texttt{ox\_flag} is set to 0.
   \texttt{ux\_flag} is set to 0.
   \texttt{xx\_flag} is set to 0.
   \texttt{zx\_flag} is set to 0.
RoundToDP(x, y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

- 0b00: Round to Nearest Even
- 0b01: Round towards Zero
- 0b10: Round towards +Infinity
- 0b11: Round towards - Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to double-precision under control of the rounding mode specified by x.

```plaintext
if IsQNaN(y) then return ConvertFPtoDP(y)
if IsInf(y) then return ConvertFPtoDP(y)
if IsZero(y) then return ConvertFPtoDP(y)
if y<Nmin then do
  if UE=0 then do
    if x=0b00 then r ← RoundToDPNearEven( DenormDP(y) )
    if x=0b01 then r ← RoundToDPT trunc( DenormDP(y) )
    if x=0b10 then r ← RoundToDPCeil( DenormDP(y) )
    if x=0b11 then r ← RoundToDPFloor( DenormDP(y) )
    ux_flag ← xx_flag
    return(ConvertFPtoDP(r))
  end
  else do
    y ← Scalb(y,+1536)
    ux_flag ← 1
  end
end
if x=0b00 then r ← RoundToDPNearEven(y)
if x=0b01 then r ← RoundToDPT trunc(y)
if x=0b10 then r ← RoundToDPCeil(y)
if x=0b11 then r ← RoundToDPFloor(y))
if r>Nmax then do
  if OE=0 then do
    if x=0b00 then r ← sign ? -Inf : +Inf
    if x=0b01 then r ← sign ? -Nmax : +Nmax
    if x=0b10 then r ← sign ? -Nmax : +Inf
    if x=0b11 then r ← sign ? -Inf : +Nmax
    ox_flag ← 0b1
    xx_flag ← 0b1
    inc_flag ← 0bU
    return(ConvertFPtoDP(r))
  end
  else do
    r ← Scalb(r,-1536)
    ox_flag ← 1
  end
end
return(ConvertFPtoDP(r))
```
RoundToDPCeil(x)
   x is a floating-point value having unbounded range and precision.
   
   If x is a QNaN, return x.
   
   Otherwise, if x is an Infinity, return x.
   
   Otherwise, do the following.
   Return the smallest floating-point number having unbounded exponent range but double-precision
   significand precision that is greater or equal in value to x.
   
   If the magnitude of the value returned is greater than x, inc_flag is set to 1.
   
   If the value returned is not equal to x, xx_flag is set to 1.
   
RoundToDPFloor(x)
   x is a floating-point value having unbounded range and precision.
   
   If x is a QNaN, return x.
   
   Otherwise, if x is an Infinity, return x.
   
   Otherwise, do the following.
   Return the largest floating-point number having unbounded exponent range but double-precision
   significand precision that is lesser or equal in value to x.
   
   If the magnitude of the value returned is greater than x, inc_flag is set to 1.
   
   If the value returned is not equal to x, xx_flag is set to 1.
   
RoundToDPIntegerCeil(x)
   x is a double-precision floating-point value.
   
   If x is an SNaN, vxsnan_flag is set to 1.
   
   If x is a QNaN, return x.
   
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   
   Otherwise, if x is an infinity, return x.
   
   Otherwise, do the following.
   Return the smallest double-precision floating-point integer value that is greater or equal in value to x.
   
   If the magnitude of the value returned is greater than x, inc_flag is set to 1.
   
   If the value returned is not equal to x, xx_flag is set to 1.
RoundToDPIntegerFloor(x)

x is a double-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is an infinity, return x.
Otherwise, do the following.
Return the largest double-precision floating-point integer value that is lesser or equal in value to x
If the magnitude of the value returned is greater than x, inc_flag is set to 1.
If the value returned is not equal to x, xx_flag is set to 1.

RoundToDPIntegerNearAway(x)

x is a double-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is an infinity, return x.
Otherwise, do the following.
Return the largest double-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest double-precision floating-point integer that is greater or equal in value to x-0.5 if x<0.
If the magnitude of the value returned is greater than x, inc_flag is set to 1.
If the value returned is not equal to x, xx_flag is set to 1.

RoundToDPIntegerNearEven(x)

x is a double-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.
If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is an infinity, return x.
Otherwise, do the following.
Return the double-precision floating-point integer value that is nearest in value to x (in case of a tie, the double-precision floating-point integer value with the least-significant bit equal to 0 is used).
If the magnitude of the value returned is greater than x, inc_flag is set to 1.
If the value returned is not equal to x, xx_flag is set to 1.
RoundToDPIntegerTrunc(x)

x is a double-precision floating-point value.

If x is an SNaN, \(vxsnan\_flag\) is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.
Return the largest double-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest double-precision floating-point integer value that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, \(inc\_flag\) is set to 1.

If the value returned is not equal to x, \(xx\_flag\) is set to 1.

RoundToDPNearEven(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.
Return the floating-point number having unbounded exponent range but double-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but double-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, \(inc\_flag\) is set to 1.

If the value returned is not equal to x, \(xx\_flag\) is set to 1.

RoundToDPTrunc(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.
Return the largest floating-point number having unbounded exponent range but double-precision significand precision that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, \(inc\_flag\) is set to 1.

If the value returned is not equal to x, \(xx\_flag\) is set to 1.
RoundToSP(x,y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

- 0b00: Round to Nearest Even
- 0b01: Round towards Zero
- 0b10: Round towards +Infinity
- 0b11: Round towards -Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to single-precision under control of the rounding mode specified by x.

```plaintext
if IsQNaN(y) then return ConvertFPtoSP(y)
if IsInf(y) then return ConvertFPtoSP(y)
if IsZero(y) then return ConvertFPtoSP(y)
if y<Nmin then do
  if UE=0 then do
    if x=0b00 then r ← RoundToSPNearEven( DenormSP(y) )
    if x=0b01 then r ← RoundToSPTrunc( DenormSP(y) )
    if x=0b10 then r ← RoundToSPCeil( DenormSP(y) )
    if x=0b11 then r ← RoundToSPFloor( DenormSP(y) )
    ux_flag ← xx_flag
    return(ConvertFPtoSP(r))
  end
  else do
    y ← Scalb(y,+192)
    ux_flag ← 1
  end
end
if x=0b00 then r ← RoundToSPNearEven(y)
if x=0b01 then r ← RoundToSPTrunc(y)
if x=0b10 then r ← RoundToSPCeil(y)
if x=0b11 then r ← RoundToSPFloor(y))
if r>Nmax then do
  if OE=0 then do
    if x=0b00 then r ← sign ? -Inf : +Inf
    if x=0b01 then r ← sign ? -Nmax : +Nmax
    if x=0b10 then r ← sign ? -Nmax : +Inf
    if x=0b11 then r ← sign ? -Inf : +Nmax
    ox_flag ← 0b1
    xx_flag ← 0b1
    inc_flag ← 0bU
    return(ConvertFPtoSP(r))
  end
  else do
    r ← Scalb(r,-192)
    ox_flag ← 1
  end
end
return(ConvertFPtoSP(r))
```
RoundToSPCeil(x)
x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.
Return the smallest floating-point number having unbounded exponent range but single-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

RoundToSPFloor(x)
x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.
Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

RoundToSPIntegerCeil(x)
x is a single-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.
Return the smallest single-precision floating-point integer value that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.
RoundToSPIntegerFloor(x)
   x is a single-precision floating-point value.
   If x is an SNaN, \(v_{x\text{snan}\_\text{flag}}\) is set to 1.
   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if x is an infinity, return x.
   Otherwise, do the following.
      Return the largest single-precision floating-point integer value that is lesser or equal in value to x.
      If the magnitude of the value returned is greater than x, \(\text{inc\_flag}\) is set to 1.
      If the value returned is not equal to x, \(xx\_\text{flag}\) is set to 1.

RoundToSPIntegerNearAway(x)
   x is a single-precision floating-point value.
   If x is an SNaN, \(v_{x\text{snan}\_\text{flag}}\) is set to 1.
   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if x is an infinity, return x.
   Otherwise, do the following.
      Return x if x is a floating-point integer; otherwise return the largest single-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x-0.5 if x<0.
      If the magnitude of the value returned is greater than x, \(\text{inc\_flag}\) is set to 1.
      If the value returned is not equal to x, \(xx\_\text{flag}\) is set to 1.

RoundToSPIntegerNearEven(x)
   x is a single-precision floating-point value.
   If x is an SNaN, \(v_{x\text{snan}\_\text{flag}}\) is set to 1.
   If x is a QNaN, return x.
   Otherwise, if x is an SNaN, return x represented as a QNaN.
   Otherwise, if x is an infinity, return x.
   Otherwise, do the following.
      Return x if x is a floating-point integer; otherwise return the single-precision floating-point integer value that is nearest in value to x (in case of a tie, the single-precision floating-point integer value with the least-significant bit equal to 0 is used).
      If the magnitude of the value returned is greater than x, \(\text{inc\_flag}\) is set to 1.
      If the value returned is not equal to x, \(xx\_\text{flag}\) is set to 1.
RoundToSPIntegerTrunc(x)
   x is a single-precision floating-point value.

   If x is a QNaN, return x.

   Otherwise, if x is an SNaN, return x represented as a QNaN, and vxsnan_flag is set to 1.

   Otherwise, if x is an infinity, return x.

   Otherwise, do the following.
   Return the largest single-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x if x<0.

   If the magnitude of the value returned is greater than x, inc_flag is set to 1.

   If the value returned is not equal to x, xx_flag is set to 1.

RoundToSPNearEven(x)
   x is a floating-point value having unbounded range and precision.

   If x is a QNaN, return x.

   Otherwise, if x is an Infinity, return x.

   Otherwise, do the following.
   Return the floating-point number having unbounded exponent range but single-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but single-precision significand precision with the least-significant bit equal to 0 is used).

   If the magnitude of the value returned is greater than x, inc_flag is set to 1.

   If the value returned is not equal to x, xx_flag is set to 1.

RoundToSPTTrunc(x)
   x is a floating-point value having unbounded range and precision.

   If x is a QNaN, return x.

   Otherwise, if x is an Infinity, return x.

   Otherwise, do the following.
   Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point number that is greater or equal in value to x if x<0.

   If the magnitude of the value returned is greater than x, inc_flag is set to 1.

   If the value returned is not equal to x, xx_flag is set to 1.

Scalb(x,y)
   x is a floating-point value having unbounded range and precision.
   y is a signed integer.

   Result of multiplying the floating-point value x by 2^y.

SetFX(x)
   x is one of the exception flags in the FPSCR.

   If the contents of x is 0, FX and x are set to 1.
SquareRootDP(x)
x is a double-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.
If x is a negative, nonzero value, vxsqrt_flag is set to 1.
If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.

SquareRootSP(x)
x is a single-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.
If x is a negative, nonzero value, vxsqrt_flag is set to 1.
If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.
7.6.3 VSX Instruction Descriptions

Load VSX Scalar Doubleword Indexed XX1-form

\[
\text{lxsdx XT,RA,RB}
\]

\[
\begin{array}{cccccc}
31 & T & RA & RB & 588 & TX \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} || T \\
a[0:63] & \leftarrow \begin{cases} 
0 & \text{if RA} = 0 \\
\text{GPR}[RA] & \text{otherwise}
\end{cases} \\
\text{EA}[0:63] & \leftarrow a + \text{GPR}[RB] \\
\text{VSR}[\text{TX}] & \leftarrow \text{MEM(EA,8)} || 0xUUUU_UUUU_UUUU_UUUU
\end{align*}
\]

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the doubleword in storage at address EA are placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for lxsdx

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM(EA,8)</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

Load VSX Scalar Doubleword Indexed XX1-form

Special Registers Altered

None

VSR Data Layout for lxsiwax

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>
**Load VSX Scalar as Integer Word and Zero Indexed XX1-form**

lxsiwzx  XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>12</th>
<th>TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The 32-bit unsigned integer value in the word in storage at address EA is zero-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered**

None

**VSR Data Layout for lxsiwzx**

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>UD</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>

---

**Load VSX Scalar Single-Precision Indexed XX1-form**

lxsspx  XT,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>524</th>
<th>TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The single-precision floating-point value in the word in storage at address EA is placed in doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered**

None

**VSR Data Layout for lxsspx**

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>0</th>
<th>DP</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>
Load VSX Vector Doubleword*2 Indexed XX1-form

\[ \text{lxvd2x}\]

<table>
<thead>
<tr>
<th>XT, RA, RB</th>
<th>31</th>
<th>T</th>
<th>RA</th>
<th>RB</th>
<th>844</th>
<th>XT</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0:63]</td>
<td>← [RA=0] ? 0 : GPR[RA]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA[0:63]</td>
<td>← a + GPR[RB]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT][0:63]</td>
<td>← MEM(EA, 8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT][64:127]</td>
<td>← MEM(EA+8, 8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the doubleword in storage at address EA are placed into doubleword element 0 of VSR[XT].

The contents of the doubleword in storage at address EA+8 are placed into doubleword element 1 of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvd2x

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>MEM(EA, 8)</th>
<th>MEM(EA+8, 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

Extended Mnemonic | Equivalent To | Usage |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lxvx</td>
<td>XT, RA, RB</td>
<td>lxvd2x XT, RA, RB can be used for vector load operations using Big-Endian byte-ordering, independent of element size</td>
</tr>
</tbody>
</table>
**Load VSX Vector Word*4 Indexed XX1-form**

<table>
<thead>
<tr>
<th>lxvw4x</th>
<th>XT,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>RA</td>
</tr>
<tr>
<td>31</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the word in storage at address EA are placed into word element 0 of VSR[XT].

The contents of the word in storage at address EA+4 are placed into word element 1 of VSR[XT].

The contents of the word in storage at address EA+8 are placed into word element 2 of VSR[XT].

The contents of the word in storage at address EA+12 are placed into word element 3 of VSR[XT].

**Special Registers Altered**

None

**VSR Data Layout for lxvw4x**

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>MEM(EA,4)</th>
<th>MEM(EA+4,4)</th>
<th>MEM(EA+8,4)</th>
<th>MEM(EA+12,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

**Store VSX Scalar Doubleword Indexed XX1-form**

<table>
<thead>
<tr>
<th>stxsdx</th>
<th>XS,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>RA</td>
</tr>
<tr>
<td>31</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of doubleword element 0 of VSR[XS] are placed in the doubleword in storage at address EA.

**Special Registers Altered**

None

**VSR Data Layout for stxsdx**

src = VSR[XS]

<table>
<thead>
<tr>
<th>DP/SD/UD/MD</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
Store VSX Scalar as Integer Word Indexed XX1-form

\[
\text{stxsiwx} \rightarrow \text{XS,RA,RB}
\]

\[
\begin{array}{cccccc}
31 & 6 & 140 & 5 & 11 & 16 & 21 & 31 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{align*}
\text{EA} & \leftarrow (\text{RA} = 0) \ ? \ 0 : \text{GPR}[\text{RA}] + \text{GPR}[\text{RB}] \\
\text{MEM} & \leftarrow \text{VSR}[\text{32} \times \text{SX} + \text{S}] . \text{word}[1]
\end{align*}
\]

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of word element 1 of VSR[XS] is placed in the word in storage at address EA in single-precision format.

Special Registers Altered

None

VSR Data Layout for stxsspx

\[
\begin{array}{cccccc}
\text{src} = \text{VSR}[\text{XS}] \\
\begin{array}{cccc}
\text{unused} & \text{SW/UW} & \text{unused} \\
0 & 32 & 64 & 127
\end{array}
\end{array}
\]

Store VSX Scalar Single-Precision Indexed XX1-form

\[
\text{stxsspx} \rightarrow \text{XS,RA,RB}
\]

\[
\begin{array}{cccccc}
31 & 6 & 140 & 5 & 11 & 16 & 21 & 31 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{align*}
\text{EA} & \leftarrow (\text{RA} = 0) \ ? \ 0 : \text{GPR}[\text{RA}] + \text{GPR}[\text{RB}] \\
\text{MEM} & \leftarrow \text{ConvertSP64toSP}(\text{VSR}[\text{32} \times \text{SX} + \text{S}] . \text{doubleword}[0])
\end{align*}
\]

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The single-precision value in double-precision floating-point format in doubleword element 0 of VSR[XS] is placed in the word in storage at address EA in single-precision format.

Special Registers Altered

None

VSR Data Layout for stxsspx

\[
\begin{array}{cccccc}
\text{src} = \text{VSR}[\text{XS}] \\
\begin{array}{cccc}
\text{DP} & \text{unused} \\
0 & 64 & 127
\end{array}
\end{array}
\]

Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 397
**Store VSX Vector Doubleword**\(^2\) Indexed \(XX1\)-form

\[
\text{stxvd2x} \quad \text{XS,RA,RB}
\]

\[
\begin{array}{cccccccc}
31 & 6 & 11 & 16 & 21 & 972 & 31 \\
\end{array}
\]

- \(\text{XS} \leftarrow \text{SX} \| \text{S}\)
- \(\text{a}(0:63) \leftarrow (\text{RA}=0) \ ? \ 0 : \text{GPR}[\text{RA}]\)
- \(\text{EA}(0:63) \leftarrow \text{a} + \text{GPR}[\text{RB}]\)
- \(\text{MEM}(\text{EA},8) \leftarrow \text{VSR}[\text{XS}](0:63)\)
- \(\text{MEM}(\text{EA}+8,8) \leftarrow \text{VSR}[\text{XS}][64:127]\)

Let \(\text{XS}\) be the value \(\text{SX}\) concatenated with \(\text{S}\).

Let \(\text{EA}\) be the sum of the contents of \(\text{GPR}[\text{RA}]\), or 0 if \(\text{RA}\) is equal to 0, and the contents of \(\text{GPR}[\text{RB}]\).

The contents of doubleword element 0 of \(\text{VSR}[\text{XS}]\) are placed in the doubleword in storage at address \(\text{EA}\).

The contents of doubleword element 1 of \(\text{VSR}[\text{XS}]\) are placed in the doubleword in storage at address \(\text{EA}+8\).

**Special Registers Altered**

None

**VSR Data Layout for stxvd2x**

\[
\begin{array}{cccc}
\text{src} = \text{VSR}[\text{XS}] \\
& \cdot \text{dword}[0] & \cdot \text{dword}[1] \\
0 & 64 & 127 \\
\end{array}
\]

**Extended Mnemonic Equivalent To**

\(\text{stxvx} \quad \text{XS,RA,RB} \quad \text{stxvd2x} \quad \text{XS,RA,RB}\)

Usage:

- can be used for vector store operations using Big-Endian byte-ordering, independent of element size

---

**Store VSX Vector Word**\(^4\) Indexed \(XX1\)-form

\[
\text{stxvw4x} \quad \text{XS,RA,RB}
\]

\[
\begin{array}{cccccccc}
31 & 6 & 11 & 16 & 21 & 908 & 31 \\
\end{array}
\]

- \(\text{XS} \leftarrow \text{SX} \| \text{S}\)
- \(\text{a}(0:63) \leftarrow (\text{RA}=0) \ ? \ 0 : \text{GPR}[\text{RA}]\)
- \(\text{EA}(0:63) \leftarrow \text{a} + \text{GPR}[\text{RB}]\)
- \(\text{MEM}(\text{EA},4) \leftarrow \text{VSR}[\text{XS}](0:31)\)
- \(\text{MEM}(\text{EA}+4,4) \leftarrow \text{VSR}[\text{XS}][32:63]\)
- \(\text{MEM}(\text{EA}+8,4) \leftarrow \text{VSR}[\text{XS}][64:95]\)
- \(\text{MEM}(\text{EA}+12,4) \leftarrow \text{VSR}[\text{XS}][96:127]\)

Let \(\text{XS}\) be the value \(\text{SX}\) concatenated with \(\text{S}\).

Let \(\text{EA}\) be the sum of the contents of \(\text{GPR}[\text{RA}]\), or 0 if \(\text{RA}\) is equal to 0, and the contents of \(\text{GPR}[\text{RB}]\).

The contents of word element 0 of \(\text{VSR}[\text{XS}]\) are placed in the word in storage at address \(\text{EA}\).

The contents of word element 1 of \(\text{VSR}[\text{XS}]\) are placed in the word in storage at address \(\text{EA}+4\).

The contents of word element 2 of \(\text{VSR}[\text{XS}]\) are placed in the word in storage at address \(\text{EA}+8\).

The contents of word element 3 of \(\text{VSR}[\text{XS}]\) are placed in the word in storage at address \(\text{EA}+12\).

**Special Registers Altered**

None

**VSR Data Layout for stxvw4x**

\[
\begin{array}{cccc}
\text{src} = \text{VSR}[\text{XS}] \\
& \cdot \text{word}[0] & \cdot \text{word}[1] & \cdot \text{word}[2] & \cdot \text{word}[3] \\
0 & 32 & 64 & 96 & 127 \\
\end{array}
\]
**VSX Scalar Absolute Value Double-Precision XX2-form**

```
xsabsdp    XT,XB
```

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>0</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>345</th>
<th>351</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>→ TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>→ BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

result[0:63] ← 0b0 || VSR[XB][1:63]
VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

The absolute value of the double-precision floating-point operand in doubleword element 0 of VSR[XB] is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered**
None

**VSR Data Layout for xsabsdp**

src = VSR[XB]

tgt = VSR[XT]

```
src = VSR[XB]
```

```
tgt = VSR[XT]
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DP</td>
<td>undefined</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VSX Scalar Add Double-Precision XX3-form

xsadddp          XT,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>56</th>
<th>51</th>
<th>16</th>
<th>21</th>
<th>32</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>V</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>XT ← IX</td>
<td></td>
<td>T</td>
<td>XA ← AX</td>
<td></td>
<td>A</td>
<td>XB ← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>src1 ← VSR<a href="">XT</a></td>
<td>src2 ← VSR<a href="">XA</a></td>
<td>v(0:inf) ← AddDP(src1,src2)</td>
<td>result(0:63) ← RoundToDP(RN,v)</td>
<td>if(vxsnan_flag) then SetFX(VXSNAN)</td>
<td>if(vxisi_flag) then SetFX(VXISI)</td>
<td>if(ox_flag) then SetFX(OX)</td>
<td>if(ux_flag) then SetFX(UX)</td>
<td>if(xx_flag) then SetFX(XX)</td>
<td>vex_flag ← VE &amp; (vxsnan_flag</td>
<td>vxisi_flag)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if(~vex_flag) then do</td>
<td>VSR[XT] ← result</td>
<td></td>
<td>0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FPRF ← ClassSP(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FR ← inc_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FI ← xx_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else do</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FR ← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FI ← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 48, “Actions for xsadddp,” on page 401.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity v ← -Infinity v ← -Infinity v ← -Infinity v ← +dQNaN vxisi_flag ← 1     v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity v ← A(src1,src2) v ← src1 v ← A(src1,src2) v ← +Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity v ← src2 v ← -Zero v ← Rezd v ← src2 v ← +Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity v ← src2 v ← Rezd v ← +Zero v ← src2 v ← +Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity v ← A(src1,src2) v ← src1 v ← A(src1,src2) v ← +Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN vxisi_flag ← 1</td>
<td>v ← +Infinity v ← +Infinity v ← +Infinity v ← +Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
- src1: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- src2: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- dQNaN: Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- A(x,y): Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
  Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- Q(x): Return a QNaN with the payload of x.
- v: The intermediate result having unbounded significand precision and unbounded exponent range.

Table 48. Actions for xsadddp
### Table 49. Floating-Point Intermediate Result Handling

<table>
<thead>
<tr>
<th>Range of v</th>
<th>Case</th>
<th>Rounding Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>v is a QNaN</td>
<td>Special</td>
<td>RTN</td>
</tr>
<tr>
<td>v = +∞</td>
<td>Special</td>
<td>v ← v</td>
</tr>
<tr>
<td>-(Nmax + 1ulp) &lt; v ≤ -(Nmax + 1ulp)</td>
<td>Overflow</td>
<td>q ← Rnd(v)</td>
</tr>
<tr>
<td>-(Nmax + 1ulp) &lt; v ≤ -(Nmax)</td>
<td>Normal</td>
<td>q ← Rnd(v)</td>
</tr>
<tr>
<td>-(Nmax + ½ulp) &lt; v &lt; -Nmax</td>
<td>Normal</td>
<td>q ← Rnd(v)</td>
</tr>
<tr>
<td>-Nmax &lt; v &lt; -Nmin</td>
<td>Normal</td>
<td>t ← +Nmax</td>
</tr>
<tr>
<td>-Nmin &lt; v &lt; -Zero</td>
<td>Tiny</td>
<td>t ← Rnd(v)</td>
</tr>
<tr>
<td>v = -Zero</td>
<td>Special</td>
<td>t ← v</td>
</tr>
<tr>
<td>v = Rezd</td>
<td>Special</td>
<td>t ← +Zero</td>
</tr>
<tr>
<td>v = +Zero</td>
<td>Special</td>
<td>t ← v</td>
</tr>
<tr>
<td>+Zero &lt; v &lt; +Nmin</td>
<td>Tiny</td>
<td>q ← Rnd(v)</td>
</tr>
<tr>
<td>v = +Nmin</td>
<td>Normal</td>
<td>t ← +Nmin</td>
</tr>
<tr>
<td>+Nmin &lt; v &lt; +Nmax</td>
<td>Normal</td>
<td>t ← +Nmax</td>
</tr>
<tr>
<td>+Nmax &lt; v &lt; +∞</td>
<td>Normal</td>
<td>t ← +Infinity</td>
</tr>
<tr>
<td>-(Nmax + 1ulp) ≤ v &lt; -(Nmax + 1ulp)</td>
<td>Overflow</td>
<td>q ← Rnd(v)</td>
</tr>
<tr>
<td>+(Nmax + 1ulp) ≤ v &lt; +∞</td>
<td>Overflow</td>
<td>q ← Rnd(v)</td>
</tr>
<tr>
<td>v = +Infinity</td>
<td>Special</td>
<td>t ← v</td>
</tr>
<tr>
<td>v = −Infinity</td>
<td>Special</td>
<td>t ← v</td>
</tr>
</tbody>
</table>

**Explanation:**
- **v** The precise intermediate result defined in the instruction having unbounded range and precision.
- **Den(x)** The value x is denormalized. The significand is shifted left by the amount of the difference between Emin for the target precision (that is, -1022 for double-precision, -126 for single-precision) and the unbiased exponent of x. The unbiased exponent of the denormalized value is Emin. The significand of the denormalized value has unbounded significand precision.
- **Rezd** Exact-zero-difference result. Applies only to add operations involving source operands having the same magnitude and different signs.
- **Rnd(x)** The significand of x is rounded to the target precision according to the rounding mode specified in FPSCR<sub>RN</sub>. Exponent range of the rounded result is unbounded. See Section 7.3.2.6.
- **Nmax** Largest (in magnitude) representable normalized number in the target precision format.
- **Nmin** Smallest (in magnitude) representable normalized number in the target precision format.
- **ulp** Least significant bit in the target precision format’s significand (Unit in the Last Position).
- **RTN** Round To Nearest, ties to Even.
- **RTZ** Round Toward Zero.
- ** RTP** Round Toward +infinity.
- **RTM** Round Toward −infinity.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

### Case

<table>
<thead>
<tr>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>vxidi_flag</th>
<th>vxzdz_flag</th>
<th>vxsqrt_flag</th>
<th>zx_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**

- The results do not depend on this condition.
- ClassFP(x) Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
- fx(x) FX is set to 1 if x=0.  x is set to 1.
- β Wrap adjust, where \( \beta = 2^{1536} \) for double-precision and \( \beta = 2^{192} \) for single-precision.
- q The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range.
- r The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range.
- v The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- FI Floating-Point Fraction Inexact status flag, FPSCRFI. This status flag is nonsticky.
- FR Floating-Point Fraction Rounded status flag, FPSCRFR.
- OX Floating-Point Overflow exception status flag, FPSCRFX.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- T(x) The value x is placed in element 0 of VSR[XT] in the target precision format.
- UX Floating-Point Underflow exception status flag, FPSCRUX.
- VXSQRT Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCRVXSQRT.
- VXISI Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCRVXISI.
- VXZDZ Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, FPSCRVXZDZ.
- XX Floating-Point Inexact Exception status flag, FPSCRXX. The flag is a sticky version of FPSCRXX. When FPSCRXX is set to a new value, the new value of FPSCRXX is set to the result of ORing the old value of FPSCRXX with the new value of FPSCRXX.
- ZX Floating-Point Zero Divide Exception status flag, FPSCRZX.

Table 50. Scalar Floating-Point Final Result
### Returned Results and Status Setting

| Case | VE | OE | UE | ZE | XE | vxsnan_flag | vexsor_flag | vxidi_flag | vxzdz_flag | vxsqrt_flag | vxz_flag | ClassFP(x) | FR (≥ v) | FR (≥ 0) | FR (≥ v) > 0 | Is q Inexact? (g ≠ v) | Is q Incremented? (g ≠ v) | Is r Incremented? (|r| > |v|) |
|------|----|----|----|----|----|-------------|-------------|------------|------------|-------------|----------|-------------|----------|----------|-----------------|-----------------|----------------------|-----------------|
| Normal | – | – | – | – | – | – | – | – | – | – | – | T(1), PPRF→ClassFP(x), FR=0, FR=0 | T(1), PPRF→ClassFP(x), FR–1, FR=0, Fx(XX) | T(1), PPRF→ClassFP(x), FR=1, FR–1, Fx(XX) | T(1), PPRF→ClassFP(x), FR=0, Fx(XX) | error() |
| Overflow | – | 0 | – | – | – | – | – | – | – | – | – | T(1), PPRF→ClassFP(x), FR=0, FR–1, Fx(XX) | T(1), PPRF→ClassFP(x), FR=0, Fx(XO), Fx(XX) | T(1), PPRF→ClassFP(x), FR=0, Fx(XO), Fx(XX) | error() |
| Tiny | – | 0 | – | – | – | – | – | – | – | – | – | T(1), PPRF→ClassFP(x), FR–1, FR=0, Fx(XX) | T(1), PPRF→ClassFP(x), FR–1, FR=0, Fx(XX) | error() |

### Explanation:

– The results do not depend on this condition.

ClassFP(x) Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.

fx(x) FX is set to 1 if x=0. x is set to 1.

β Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{1022}$ for single-precision.

q The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range.

r The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range.

v The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.

Fi Floating-Point Fraction Inexact status flag, FPSCR$_{Fx}$. This status flag is nonsticky.

FR Floating-Point Fraction Rounded status flag, FPSCR$_{FR}$.

OX Floating-Point Overflow exception status flag, FPSCR$_{OX}$.

error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.

T(x) The value x is placed in element 0 of VSR[$X$T] in the target precision format.

 Returned Results and Status Setting | Returned Results and Status Setting | Returned Results and Status Setting | Returned Results and Status Setting | Returned Results and Status Setting | Returned Results and Status Setting | Returned Results and Status Setting | Returned Results and Status Setting |

Table 50. Scalar Floating-Point Final Result (Continued)
**VSX Scalar Add Single-Precision XX3-form**

xsaddsp       XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>0</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

reset_xflags()

src1 ← VSR[32×AX+A].doubleword[0]
src2 ← VSR[32×BX+B].doubleword[0]

v ← AddDP(src1,src2)

result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxisi_flag)

if(~vex_flag) then do
  VSR[32×TX+T].doubleword[0] ← ConverSPtoDP(result)
  VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
  FPRF ← ClassSP(result)
  FR ← inc_flag
  FI ← xx_flag
  end
else do
  FR ← 0b0
  FI ← 0b0
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added\(^1\) to src1, producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 51, “Actions for xsaddsp,” on page 406.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **A(x,y)**: Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
- **Note**: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

**Table 51. Actions for xsaddsp**
VSX Scalar Compare Ordered
Double-Precision XX3-form

\[ xscmpodp \rightarrow BF, XA, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>//</th>
<th>A</th>
<th>B</th>
<th>43</th>
<th>//</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>30</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>16</td>
<td>23</td>
</tr>
</tbody>
</table>

Let \( XA \) be the value \( AX || A \).
Let \( XB \) be the value \( BX || B \).

reset_xflags()

\[
\begin{align*}
\text{src1} & \leftarrow \text{VSR}[XA][0:63] \\
\text{src2} & \leftarrow \text{VSR}[XB][0:63]
\end{align*}
\]

if( IsSNaN(src1) | IsSNaN(src2) ) then do
  vxsnan_flag \leftarrow 0b1
  if(VE=0) then vxvc_flag \leftarrow 0b1
end
else if( IsQNaN(src1) | IsQNaN(src2) ) then vxvc_flag \leftarrow 0b1

FL \leftarrow \text{CompareLTDP(src1,src2)}
FG \leftarrow \text{CompareGTDP(src1,src2)}
FE \leftarrow \text{CompareEQDP(src1,src2)}
FU \leftarrow \text{IsNAN(src1)} \text{|| IsNAN(src2)}
CR[BF] \leftarrow FL \text{|| FG \text{|| FE \text{|| FU}}}
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxvc_flag) then SetFX(VXVC)

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal.

Infinities of same signs compare equal.

See Table 52, “Actions for xscmpodp - Part 1: Compare Ordered,” on page 408.

The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set, and Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, VXVC is set.

See Table 53, “Actions for xscmpodp - Part 2: Result,” on page 408.
Table 52. Actions for xcmpdp - Part 1: Compare Ordered

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
</tr>
<tr>
<td>–NZF</td>
<td>cc=0b0100</td>
<td>cc=C(src1,src2)</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b1000</td>
<td>cc=0b0001</td>
<td>vxvc_flag=1</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
</tr>
<tr>
<td>–Zero</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b0010</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b0001</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
</tr>
<tr>
<td>+Zero</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b0010</td>
<td>cc=0b0010</td>
<td>cc=0b1000</td>
<td>cc=0b0001</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0100</td>
<td>cc=0b0010</td>
<td>cc=0b0001</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
</tr>
<tr>
<td>QNaN</td>
<td>cc=0b0001 vxvc_flag=1</td>
<td>cc=0b0001 vxvc_flag=1</td>
<td>cc=0b0001 vxvc_flag=1</td>
<td>cc=0b0001 vxvc_flag=1</td>
<td>cc=0b0001 vxvc_flag=1</td>
<td>cc=0b0001 vxvc_flag=1</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td>cc=0b0001 vxsnan_flag=1 vxvc_flag=1 (VE=0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
src1 The double-precision floating-point value in doubleword element 0 of VSR[XA];
src2 The double-precision floating-point value in doubleword element 0 of VSR[XB];
NZF Nonzero finite number.
C(x,y) The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results.
  0b1000 when x is less than y
  0b0100 when x is equal to y
  0b0010 when x is greater than y
cc The 4-bit result compare code.

Table 53. Actions for xcmpdp - Part 2: Result

<table>
<thead>
<tr>
<th>VE</th>
<th>Vxsnan_flag</th>
<th>Vxvc_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>FPCC=&lt;cc, CR(BF)=&lt;cc&gt;</td>
</tr>
<tr>
<td>–</td>
<td>0 0</td>
<td></td>
<td>FPCC=&lt;cc, CR(BF)=&lt;cc, fx(VXVC)&gt;</td>
</tr>
<tr>
<td>0</td>
<td>0 1</td>
<td></td>
<td>FPCC=&lt;cc, CR(BF)=&lt;cc, fx(VXSNAN)&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1 1</td>
<td></td>
<td>FPCC=&lt;cc, CR(BF)=&lt;cc, fx(VXSNAN), fx(VXVC)&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
<td></td>
<td>FPCC=&lt;cc, CR(BF)=&lt;cc, fx(VXVC), error()&gt;</td>
</tr>
<tr>
<td>1</td>
<td>1 –</td>
<td></td>
<td>FPCC=&lt;cc, CR(BF)=&lt;cc, fx(VXSNAN), error()&gt;</td>
</tr>
</tbody>
</table>

Explanation:
– The results do not depend on this condition.
cc The 4-bit result as defined in Table 52.
fX(x) FX is set to 1 if x=0, x is set to 1.
error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
FX Floating-Point Summary Exception status flag, FPSCRFX
VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCRVXSNAN See Section 7.4.1.
VXC Floating-Point Invalid Operation Exception (Invalid Compare) status flag, FPSCRVXVC See Section 7.4.1.
**VSX Scalar Compare Unordered Double-Precision XX3-form**

xscmpudp  BF,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

XA  \rightarrow  AX || A
XB  \rightarrow  BX || B
reset_xflags()
src1  \leftarrow  VSR[XA][0:63]
src2  \leftarrow  VSR[XB][0:63]

if( IsSNaN(src1) | IsSNaN(src2) ) then vxsnan_flag  \leftarrow  1

FL  \leftarrow  CompareLTDP(src1,src2)
FG  \leftarrow  CompareGTDP(src1,src2)
FE  \leftarrow  CompareEQDP(src1,src2)
FU  \leftarrow  IsNAN(src1) | IsNAN(src2)
CR[BF]  \leftarrow  FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal equal.

Infinities of same signs compare equal.


The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set.


**Special Registers Altered**
CR[BF]  
FPCC  
FX  
VXSNAN
Table 54. Actions for xscmpudp - Part 1: Compare Unordered

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>cc = 0b0010</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–NZF</td>
<td>cc = 0b0100</td>
<td>cc = C(src1,src2)</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–Zero</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b1000</td>
<td>cc = 0b1000</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = C(src1,src2)</td>
<td>cc = 0b1000</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0100</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td>cc = 0b0001 vxsnan_flag = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

- src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
- src2 The double-precision floating-point value in doubleword element 0 of VSR[XB].
- NZF Nonzero finite number.
- C(x,y) The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results:
  - 0b1000 when x is greater than y
  - 0b0100 when x is less than y
  - 0b0010 when x is equal to y
- cc The 4-bit result compare code.

Table 55. Actions for xscmpudp - Part 2: Result

<table>
<thead>
<tr>
<th>VE</th>
<th>vxsnan_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>– 0</td>
<td>0</td>
<td>FPCC ← cc, CR(BF) ← cc</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>FPCC ← cc, CR(BF) ← cc, fx(VXSNAN)</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>FPCC ← cc, CR(BF) ← cc, fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

Explanation:

- The results do not depend on this condition.
- cc The 4-bit result as defined in Table 54.
- fx(x) FX is set to 1 if x=0, x is set to 1.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- FX Floating-Point Summary Exception status flag, FPSCR_FX.
- VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR_VXSNAN. See Section 7.4.1.
### VSX Scalar Copy Sign Double-Precision

**XX3-form**

#### xscpsgndp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>176</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX | T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX | A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX | B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result{0:63}</td>
<td>← VSR[XA]{0} | VSR[XB]{1:63}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT]</td>
<td>← result | 0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Bit 0 of VSR[XT] is set to the contents of bit 0 of VSR[XA].

Bits 1:63 of VSR[XT] are set to the contents of bits 1:63 of VSR[XB].

The contents of doubleword element 1 of VSR[XT] are undefined.

### Special Registers Altered

None

### VSR Data Layout for xscpsgndp

**src1 = VSR[XA]**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DP</td>
<td>unused</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**src2 = VSR[XB]**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DP</td>
<td>unused</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**tgt = VSR[XT]**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DP</td>
<td>undefined</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
VSX Scalar round Double-Precision to single-precision and Convert to Single-Precision format XX2-form

xscvdpsp          XT, XB

 fundraising
 src ← VSR[32×XB+6].dword(0)
 result ← ConvertDPtoSP(src)
 if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
 if(xx_flag) then SetFX(FPSCR.XX)
 if(ox_flag) then SetFX(FPSCR.OX)
 if(ux_flag) then SetFX(FPSCR.UX)
 vex_flag ← FPSCR.VE & vxsnan_flag
 if( ~vex_flag ) then do
 VSR[32×TX+T].word[0] ← result
 VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
 VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
 VSR[32×TX+T].word[3] ← 0xUUUU_UUUU
 FPSCR.FPRF ← ClassSP(result)
 FPSCR.FR ← inc_flag
 FPSCR.FI ← xx_flag
 end
 else do
 FPSCR.FR ← 0b0
 FPSCR.FI ← 0b0
 end

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a SNaN, the result is src converted to a QNaN (i.e., bit 12 of src is set to 1). VXSNAN is set to 1.

Otherwise, if src is a QNaN, an Infinity, or a Zero, the result is src.

Otherwise, the result is src rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element 0 of VSR[XT] in single-precision format.

The contents of word elements 1, 2, and 3 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.
VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form

\[ xscvdpspn \quad XT,XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ /</th>
<th>B</th>
<th>267</th>
<th>3XT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>3031</td>
</tr>
</tbody>
</table>

reset_xflags()
src ← VSR[32×BX+B].dword[0]
result ← ConvertDPtoSP_NS(src)
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
VSR[32×TX+T].word[3] ← 0xUUUU_UUUU

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

Let \( src \) be the single-precision floating-point value in doubleword element 0 of \( VSR[XB] \) represented in double-precision format.

\( src \) is placed into word element 0 of \( VSR[XT] \) in single-precision format.

The contents of word elements 1, 2, and 3 of \( VSR[XT] \) are undefined.

Special Registers Altered
None

VSX Data Layout for \( xscvdpspn \)

\[ src = VSR[XB] \]

\[ tgt = VSR[XT] \]

---

Programming Note

\( xscvdpsp \) should be used to convert a scalar double-precision value to vector single-precision format.

\( xscvdpspn \) should be used to convert a scalar single-precision value to vector single-precision format.

VSX Scalar truncate Double-Precision to integer and Convert to Signed Integer Doubleword format with Saturate XX2-form

\[ xscvdpsxds \quad XT,XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ /</th>
<th>B</th>
<th>344</th>
<th>37X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>3031</td>
</tr>
</tbody>
</table>

XT ← TX || T
XB ← BX || B
inc_flag ← 0b0
reset_xflags()
rnd(0:63) ← RoundDPIntegerTrunc(VSR[XB]{0:63})
result(0:63) ← ConvertDPtoSD(rnd)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag) then SetFX(VXCVI)
if(xx_flag) then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxcvi_flag)
if( ~vex_flag ) then do
VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
FPRF ← 0bUUUUU
FR ← inc_flag
FI ← xx_flag
end
else do
FR ← 0b0
FI ← 0b0
end

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

Let \( src \) be the double-precision floating-point value in doubleword element 0 of \( VSR[XB] \).

If \( src \) is a NaN, the result is the value \( 0x8000_0000_0000_0000 \) and \( VXCVI \) is set to 1. If \( src \) is an SNaN, \( VXSNAN \) is also set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \( 2^{61} - 1 \), the result is \( 0x7FFF_FFFF_FFFF_FFFF \) and \( VXCVI \) is set to 1.

Otherwise, if the rounded value is less than \( -2^{61} \), the result is \( 0x8000_0000_0000_0000 \) and \( VXCVI \) is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

If a trap-enabled invalid operation exception occurs,
− \( VSR[XT] \) and \( FPRF \) are not modified
− \( FR \) and \( FI \) are set to 0.
Otherwise,

- The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 56.

**Special Registers Altered**

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXCVI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bUUUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VSR Data Layout for xscvdpsxds**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>undefined</td>
</tr>
</tbody>
</table>

---

**Programming Note**

*xscvdpsxds* rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including *xsrdpic* which uses the rounding mode specified by the RN field in the FPSCR.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), FR=0, Fi=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fi=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td></td>
<td>T(Nmin), FR=0, Fi=1, fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td></td>
<td>T(Nmin), FR=0, Fi=1, fx(XX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmin</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmin</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), FR=0, Fi=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin &lt; src &lt; Nmax</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td></td>
<td>T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR=0, Fi=0</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td></td>
<td>T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR=0, Fi=1, fx(XX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmax</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmax</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), FR=0, Fi=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td></td>
<td>T(Nmax), FR=0, Fi=1, fx(XX)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax+1</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmax), FR=0, Fi=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fi=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a QNaN</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), FR=0, Fi=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fi=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a SNaN</th>
<th>VE</th>
<th>XE</th>
<th>Inexact? (RoundToDPintegerTrunc(src) = src)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), FR=0, Fi=0, fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, Fi=0, fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Nmin**: The smallest signed integer doubleword value, \(-2^{63} \times 0x8000_0000_0000_0000\).
- **Nmax**: The largest signed integer doubleword value, \(2^{63}-1 \times 0x7FFF_FFFF_FFFF_FFFF\).
- **src**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **T(x)**: The signed integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

### Table 56. Actions for xscvdpxsdxs
VSX Scalar truncate Double-Precision to Integer and Convert to Signed Integer Word format with Saturate XX2-form

```
xscvdpsxws   XT, XB

<table>
<thead>
<tr>
<th>60</th>
<th>56</th>
<th>51</th>
<th>46</th>
<th>41</th>
<th>88</th>
<th>81</th>
<th>72</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←  TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←  BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inc_flag ←  0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd[0:63] ← RoundToIntegerTrunc(VSR[XB][0:63])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result[0:31] ← ConvertDPtoSN(rnd)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(vxsnan_flag) then SetFX(VXSNAN)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(vxcvi_flag) then SetFX(VXCVI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(xx_flag) then SetFX(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vex_flag ← VE &amp; (vxsnan_flag</td>
<td>vxcvi_flag)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(-vex_flag ) then do</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT] ← 0xUUUU_UUUU</td>
<td></td>
<td>result</td>
<td></td>
<td>0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPRF ← 0bUUUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR ← inc_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI ← xx_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else do</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR ← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI ← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in
doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000
and VXCVI is set to 1. If src is an SNaN, VXSNAN is
also set to 1.

Otherwise, src is rounded to a floating-point integer
using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{31-1}, the result is
0x7FFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the
result is 0x8000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to
32-bit signed-integer format.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 57.

Special Registers Altered

- FPRF=0bUUUUU
- FR  FI  FX  XX
- VXSNAN  VXCVI

VSR Data Layout for xscvdpsxws

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

```
tgt = VSR[XT]
```

<table>
<thead>
<tr>
<th>undefined</th>
<th>SW</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

Programming Note

xscvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including xsrdpic which uses the rounding mode specified by the RN field in the FPSCR.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
<td>--</td>
<td>T(Nmin), FR=0, Fl=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
<td>FR=0, Fl=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin-1 &lt; src &lt; Nmin</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td>--</td>
<td>T(Nmin), FR=0, Fl=1, fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>--</td>
<td>T(Nmin), FR=0, Fl=1, fx(XX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmin</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>no</td>
<td>T(Nmin), FR=0, Fl=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmin &lt; src &lt; Nmax</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td>--</td>
<td>T(ConvertDpToSW(RoundToDpIntegerTrunc(src))), FR=0, Fl=0</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>--</td>
<td>T(ConvertDpToSW(RoundToDpIntegerTrunc(src))), FR=0, Fl=1, fx(XX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src = Nmax</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>no</td>
<td>--</td>
<td>T(Nmax), FR=0, Fl=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td>--</td>
<td>T(Nmax), FR=0, Fl=1, fx(XX)</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>--</td>
<td>T(Nmax), FR=0, Fl=1, fx(XX), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax+1</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
<td>--</td>
<td>T(Nmax), FR=0, Fl=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
<td>FR=0, Fl=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a QNaN</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
<td>--</td>
<td>T(Nmin), FR=0, Fl=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
<td>FR=0, Fl=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a SNaN</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
<td>--</td>
<td>T(Nmin), FR=0, Fl=0, fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>--</td>
<td>FR=0, Fl=0, fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Nmin**: The smallest signed integer word value, \(-2^{31}(0x8000_0000)\).
- **Nmax**: The largest signed integer word value, \(2^{31}-1 (0x7FFF_FFFF)\).
- **src**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **T(x)**: The signed integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

**Table 57. Actions for xscvdpxsxws**
VSX Scalar Truncate Double-Precision Integer and Convert to Unsigned Integer Doubleword Format with Saturate XX2-Form

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 58.

Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bUUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VXSNAN</td>
<td>VXCVI</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSR Data Layout for xscvdpuxds

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>UD</td>
</tr>
</tbody>
</table>

Programming Note

xscvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including xsrdpic which uses the rounding mode specified by the RN field in the FPSCR.
# Returned Results and Status Setting

| src ≤ Nmin | 0 | – | T(Nmin), FR=0, FI=0, fx(VXCVI) |
| src ≤ Nmin | 1 | – | FR=0, FI=0, fx(VXCVI), error() |
| Nmin < src < Nmin | 0 | yes | T(Nmin), FR=0, FI=0, fx(XX) |
| Nmin < src < Nmin | 1 | yes | T(Nmin), FR=0, FI=0, fx(XX), error() |
| src = Nmin | – | – | no |
| Nmin < src < Nmax | – | – | no |
| Nmax < src < Nmax+1 | – | – | no |
| src ≥ Nmax+1 | 0 | – | T(Nax), FR=0, FI=0, fx(VXCVI) |
| src ≥ Nmax+1 | 1 | – | FR=0, FI=0, fx(VXCVI), error() |
| src is a QNaN | 0 | – | T(Nmin), FR=0, FI=0, fx(VXCVI) |
| src is a QNaN | 1 | – | FR=0, FI=0, fx(VXCVI), error() |
| src is a SNaN | 0 | – | T(Nmin), FR=0, FI=0, fx(VXCVI), fx(VXSNAN) |
| src is a SNaN | 1 | – | FR=0, FI=0, fx(VXCVI), fx(VXSNAN), error() |

**Explanation:**

- `fx(x)`: FX is set to 1 if x=0. x is set to 1.
- `error()`: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- `Nmin`: The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
- `Nmax`: The largest unsigned integer doubleword value, 2^{64}−1 (0x7FFF_FFFF_FFFF_FFFF).
- `src`: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- `T(x)`: The unsigned integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Table 58. Actions for xscvdpuxsds
VSX Scalar truncate Double-Precision to Integer and Convert to Unsigned Integer Word format with Saturate XX2-form

```
xscvdpuxws      XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>72</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←</td>
<td>TX</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>XB</td>
<td>←</td>
<td>BX</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>inc_flag</td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd(0:63)</td>
<td>←</td>
<td>RoundToIntegerTrunc(VSR[XB][0:63])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result(0:31)</td>
<td>←</td>
<td>ConvertDpToUn(rnd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(vxsnan_flag) then SetFX(VXSNAN)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(vxcvi_flag) then SetFX(VXCVI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(xx_flag) then SetFX(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vex_flag</td>
<td>←</td>
<td>VE &amp; (vxsnan_flag</td>
<td>vxcvi_flag)</td>
<td></td>
</tr>
<tr>
<td>if(~vex_flag ) then do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT]</td>
<td>←</td>
<td>0xUUUU_UUUU</td>
<td></td>
<td>result</td>
</tr>
<tr>
<td>FPRF</td>
<td>←</td>
<td>0bUUUUU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR</td>
<td>←</td>
<td>inc_flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI</td>
<td>←</td>
<td>xx_flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR</td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI</td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{32} - 1$, the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

If a trap-enabled invalid operation exception occurs,

- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

See Table 59.

Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0bUUUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VSR Data Layout for xscvdpuxws

```
src = VSR[XB]
tgt = VSR[XT]
```

Programming Note

xscvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a `Round to Double-Precision Integer` instruction that corresponds to the desired rounding mode, including `xsrdpic` which uses the rounding mode specified by the RN field in the FPSCR.
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src $\leq$ Nmin-1</th>
<th>VE</th>
<th>XE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), FR=0, FX=0, fx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>FR=0, FX=0, fx(VXCVI), error()</td>
</tr>
</tbody>
</table>

| Nmin-1 < src < Nmin  | -  | -  | T(Nmin), FR=0, FX=1, fx(XX)         |

| src = Nmin           | -  | -  | T(Nmin), FR=0, FX=0, fx(VXCVI)      |

| Nmin < src < Nmax    | -  | -  | T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR=0, FX=0 |
|                      | 0  | yes| T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR=0, FX=1, fx(XX), error() |
|                      | 1  | yes| T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR=0, FX=1, fx(XX), error() |

| src = Nmax           | -  | -  | T(Nmax), FR=0, FX=0, fx(VXCVI)      |

| Nmax < src < Nmax+1  | -  | -  | T(Nmax), FR=0, FX=1, fx(XX)         |
|                      | 0  | yes| T(Nmax), FR=0, FX=1, fx(XX), error() |
|                      | 1  | yes| T(Nmax), FR=0, FX=1, fx(XX), error() |

| src $\geq$ Nmax+1   | -  | -  | T(Nmax), FR=0, FX=0, fx(VXCVI)      |

| src is a QNaN        | -  | -  | T(Nmax), FR=0, FX=0, fx(VXCVI)      |
|                      | 0  | yes| FR=0, FX=0, fx(VXCVI), error()      |
|                      | 1  | yes| FR=0, FX=0, fx(VXCVI), error()      |

| src is a SNaN        | -  | -  | T(Nmax), FR=0, FX=0, fx(VXCVI)      |
|                      | 0  | -  | FR=0, FX=0, fx(VXCVI), error()      |
|                      | 1  | -  | FR=0, FX=0, fx(VXCVI), error()      |

### Table 59. Actions for xscvdpuxws

- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Nmin**: The smallest unsigned integer word value, 0 (0x0000_0000).
- **Nmax**: The largest unsigned integer word value, $2^{32} - 1$ (0xFFFF_FFFF).
- **src**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **T(x)**: The unsigned integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
VSX Scalar Convert Single-Precision to Double-Precision format **XX2-form**

**xscvspdp**

```
src ← VSR[32×BX+B].word[0]
result ← ConvertVectorSPtoScalarSP(src)
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
    vex_flag ← FPSCR.VE & vxsnan_flag
    FPSCR.FR ← 0b0
    FPSCR.FI ← 0b0
if( ~vex_flag ) then do
    VSR[32×TX+T].dword[0] ← result
    VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
    FPSCR.FPRF ← ClassDP(result)
end
```

Let $XT$ be the value $TX$ concatenated with $T$.
Let $XB$ be the value $BX$ concatenated with $B$.

Let $src$ be the single-precision floating-point value in word element 0 of $VSR[XB]$.

If $src$ is a SNaN, the result is $src$, converted to a QNaN (i.e., bit 9 of $src$ set to 1). $VXSNAN$ is set to 1.

Otherwise, the result is $src$.

The result is placed into doubleword element 0 of $VSR[XT]$ in double-precision format.

The contents of doubleword element 1 of $VSR[XT]$ are undefined.

$FPRF$ is set to the class and sign of the result. $FR$ is set to 0. $FI$ is set to 0.

If a trap-enabled invalid operation exception occurs, $VSR[XT]$ is not modified, $FPRF$ is not modified, $FR$ is set to 0, and $FI$ is set to 0.

**Special Registers Altered**

- $FPRF$
- $FR=0b0$
- $FI=0b0$
- $FX$ $VXSNAN$

**VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling**

**xscvspdpn**

```
src ← VSR[32×BX+B].word[0]
result ← ConvertVectorSPtoScalarSP(src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let $XT$ be the value $TX$ concatenated with $T$.
Let $XB$ be the value $BX$ concatenated with $B$.

Let $src$ be the single-precision floating-point value in word element 0 of $VSR[XB]$.

$src$ is placed into doubleword element 0 of $VSR[XT]$ in double-precision format.

The contents of doubleword element 1 of $VSR[XT]$ are undefined.

**Special Registers Altered**

None

**VSR Data Layout for xscvspdpn**

```
src = VSR[XB]
 tgt = VSR[XT]
```

<table>
<thead>
<tr>
<th>.word[0]</th>
<th>unused</th>
<th>unused</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

**Programming Note**

*xscvspdp* should be used to convert a single-precision floating-point value to scalar double-precision format.

*xscvspdpn* should be used to convert a vector single-precision floating-point value to scalar single-precision format.
VSX Scalar Convert Signed Integer
Doubleword to floating-point format and round to Double-Precision format XX2-form

xscvsxddp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ / /</th>
<th>B</th>
<th>376</th>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v(0:63)</td>
<td>← ConvertSDtoFP(VSR[XB].doubleword[0:63])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result(0:63)</td>
<td>← RoundToDFP(RN,v)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT]</td>
<td>← result</td>
<td></td>
<td>0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(xx_flag) then SetFX(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPRF</td>
<td>← ClassDF(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR</td>
<td>← inc_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI</td>
<td>← xx_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the signed integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered
FPRF FR FI FX XX

VSR Data Layout for xscvsxddp
src = VSR[XB]

<table>
<thead>
<tr>
<th>SD</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
<td></td>
</tr>
<tr>
<td>DP</td>
<td>undefined</td>
</tr>
</tbody>
</table>

VSX Scalar Convert Signed Integer
Doubleword to floating-point format and round to Single-Precision XX2-form

xscvsxdsp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ / /</th>
<th>B</th>
<th>312</th>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src</td>
<td>← ConvertSDtoFP(VSR[32×BX+B].doubleword[0])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td>← RoundToSP(RN,src)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[32×TX+T].doubleword[0]</td>
<td>← ConvertSsToFP64(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[32×TX+T].doubleword[1]</td>
<td>← 0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(xx_flag) then SetFX(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPRF</td>
<td>← ClassSP(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR</td>
<td>← inc_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI</td>
<td>← xx_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the two's-complement integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded.
FI is set to indicate the result is inexact.

Special Registers Altered
FPRF FR FI FX XX

VSR Data Layout for xscvsxdsp
src = VSR[XB]

<table>
<thead>
<tr>
<th>SD</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
<td></td>
</tr>
<tr>
<td>DP</td>
<td>undefined</td>
</tr>
</tbody>
</table>
VSX Scalar Convert Unsigned Integer
Doubleword to floating-point format and round to Double-Precision format XX2-form

xscvuxddp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>360</th>
<th>B X (B X)</th>
</tr>
</thead>
</table>
| XT | ← TX || T
| XB | ← BX || B
| reset_xflags() |
| src(0:inf) ← ConvertUDtoFP(VSR[XB](0:63)) |
| result(0:63) ← RoundToDP(RN,src) |
| VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU |
| if(xx_flag) then SetFX(XX) |
| FPRF ← ClassDP(result) |
| FR ← inc_flag |
| FI ← xx_flag |

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the unsigned integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

| FPRF | FR | FI | FX | XX |

VSR Data Layout for xscvuxddp

src = VSR[XB]

<table>
<thead>
<tr>
<th>UD</th>
<th>unused</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>

VSX Scalar Convert Unsigned Integer
Doubleword to floating-point format and round to Single-Precision XX2-form

xscvuxdsp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>16</th>
<th>21</th>
<th>296</th>
<th>B X (B X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src ← ConvertUDtoDP(VSR[32×BX+B].doubleword[0])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result ← RoundToSP(RN,src)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[32×TX+T].doubleword[0] ← ConvertFPtoFP64(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(xx_flag) then SetFX(XX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPRF ← ClassSP(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR ← inc_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI ← xx_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the unsigned-integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

| FPRF | FR | FI | FX | XX |

VSR Data Layout for xscvuxdsp

src = VSR[XB]

<table>
<thead>
<tr>
<th>UD</th>
<th>unused</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>
**VSX Scalar Divide Double-Precision XX3-form**

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

**Special Registers Altered**

FPRF FR FI FX OX UX ZX XX

VXSNAN VXIDI VXZDZ

**VSR Data Layout for xsdivdp**

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
<tr>
<td>0</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided\(^1\) by src2, producing a quotient having unbounded range and precision.

The quotient is normalized\(^2\).

See Actions for xsdivdp (p. 426).

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 60. Actions for xsdivdp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +NZF</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

src1  The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2  The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN  Default quiet NaN (0x7FF8_0000_0000_0000).NZF  Nonzero finite number.D(x,y)  Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.Q(x)  Return a QNaN with the payload of x.  v  The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Divide Single-Precision XX3-form

**xsdvsp** XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

reset_xflags()

src1 ← VSR[32×AX+A].doubleword[0]
src2 ← VSR[32×BX+B].doubleword[0]
v ← DivideDP(src1,src2)
result ← RoundToSP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag)  then SetFX(VXIDI)
if(vxzdz_flag)  then SetFX(VXZDZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag|vxidi_flag|vxzdz_flag)
zex_flag ← ZE & zx_flag
if( ~vex_flag & ~zex_flag ) then do
  VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
  VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
  FPRF ← ClassSP(result)
  FR ← inc_flag
  FI ← xx_flag
end
else do
  FR ← 0b0
  FI ← 0b0
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided\(^1\) by src2, producing a quotient having unbounded range and precision.

The quotient is normalized\(^2\).

See Table 61, “Actions for xsdvsp,” on page 428.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 61. Actions for xsdivsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← D(src1, src2)</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **D(x,y)**: Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **D(x,y)**: Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.
## VSX Scalar Multiply-Add Double-Precision XX3-form

For \textbf{xsmaddadp}, do the following.
- Let \textit{src2} be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let \textit{src3} be the double-precision floating-point value in doubleword element 0 of VSR[XT].

\textit{src1} is multiplied\footnote{Floating-point multiplication is based on exponent addition and multiplication of the significands.} by \textit{src3}, producing a product having unbounded range and precision.

See part 1 of Table 62.

\textit{src2} is added\footnote{Floating-point addition is based on exponent comparison and addition of the significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.} to the product, producing a sum having unbounded range and precision.

The sum is normalized\footnote{Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.}.

See part 2 of Table 62.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

### Special Registers Altered

<table>
<thead>
<tr>
<th>FPSCR</th>
<th>FX</th>
<th>UX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXISI</th>
<th>VXIMZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRF</td>
<td>FR</td>
<td>FI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xsmadd(alm)dp

- **src1** = VSR[XA]
  - |   | unused |
  - | DP |       |
- **src2** = *xsmaddadp* ? VSR[XT] : VSR[XB]
  - |   | unused |
  - | DP |       |
- **src3** = *xsmaddadp* ? VSR[XB] : VSR[XT]
  - |   | unused |
  - | DP |       |
- **tgt** = VSR[XT]
  - |   | undefined |
  - | DP |       |

0 64 127
### Table 62. Actions for xsmadd(aim)dp

<table>
<thead>
<tr>
<th>src3</th>
<th>$\infty$</th>
<th>$-\infty$</th>
<th>$0$</th>
<th>$+\infty$</th>
<th>$+\infty$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
</tr>
<tr>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
</tr>
<tr>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
</tr>
<tr>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow +\infty$</td>
</tr>
<tr>
<td>SNaN</td>
<td>$p \leftarrow -\text{snan}_1$</td>
<td>$p \leftarrow -\text{snan}_1$</td>
<td>$p \leftarrow -\text{snan}_1$</td>
<td>$p \leftarrow -\text{snan}_1$</td>
<td>$p \leftarrow -\text{snan}_1$</td>
<td>$p \leftarrow -\text{snan}_1$</td>
<td>$p \leftarrow -\text{snan}_1$</td>
</tr>
</tbody>
</table>

**Explanation:**

src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2 For xsmadddp, the double-precision floating-point value in doubleword element 0 of VSR[XT].
src3 For xsmadddp, the double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN Default quiet NaN ($\text{x07FF_0000_0000_0000}$).
NZF Nonzero finite number.
Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x) Return the normalized payload of NaN(x).
A(x,y) Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
Note: If $x = -y$, v is considered to be an exact-zero-difference result (Rezd).
M(x,y) Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p The intermediate product having unbounded range and precision.
v The intermediate result having unbounded range and precision.
VSX Scalar Multiply-Add Single-Precision
XX3-form

\[
\text{xsmaddasp} \quad \text{XT,XA, XB}
\]

\[
\begin{array}{cccc|c}
0 & 6 & 11 & 16 & 21 & 9 \\
\hline
\end{array}
\]

\[
\text{xsmaddmsp} \quad \text{XT,XA, XB}
\]

\[
\begin{array}{cccc|c}
0 & 6 & 11 & 16 & 21 & 9 \\
\hline
\end{array}
\]

reset_xflags();

if 'xsmaddasp' then do
src1 ← VSR[32×AX+A].doubleword[0]
src2 ← VSR[32×TX+T].doubleword[0]
src3 ← VSR[32×BX+B].doubleword[0]
end

if 'xsmaddmsp' then do
src1 ← VSR[32×AX+A].doubleword[0]
src2 ← VSR[32×BX+B].doubleword[0]
src3 ← VSR[32×TX+T].doubleword[0]
end

v ← MultiplyAddDP(src1, src3, src2)
result ← RoundToSP(RN, v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag) then SetFX(VXIMZ)
if(vxisi_flag) then SetFX(VXISI)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if(-vex_flag) then do
VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
FPRF ← ClassSP(result)
FR ← inc_flag
FI ← xx_flag
end
else do
FR ← 0b0
FI ← 0b0
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For \text{xsmaddasp}, do the following.

- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[AX].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[TX].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[BX].

For \text{xsmaddmsp}, do the following.

- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[AX].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[BX].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[TX].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 63, “Actions for xsmadd(a|m)sp,” on page 434.

src2 is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 63, “Actions for xsmadd(a|m)sp,” on page 434.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
Special Registers Altered

| FPRF | FR | FI | FX | OX | UX | XX | VXSNAN | VXISI | VXIMZ |

VSR Data Layout for xsmadd(a|m)sp

- src1 = VSR[XA]
  - DP: unused

- src2 = xsmaddasp ? VSR[XT] : VSR[XB]
  - DP: unused

- src3 = xsmaddasp ? VSR[XB] : VSR[XT]
  - DP: unused

- tgt = VSR[XT]
  - DP: undefined
### Table 63. Actions for xsmadd(a|m)sp

<table>
<thead>
<tr>
<th>src3</th>
<th>src2</th>
<th>src1</th>
<th>Multiply</th>
<th>Add</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-NZF</td>
<td>Zero</td>
<td>+NZF</td>
<td>+Infinity</td>
<td>QNaN</td>
</tr>
<tr>
<td>p←+Infinity</td>
<td>p←+Infinity</td>
<td>p←+dQNaN</td>
<td>p←+Infinity</td>
<td>p←-Infinity</td>
<td>p←src3</td>
</tr>
<tr>
<td>p←+Infinity</td>
<td>p←M(src1,src3)</td>
<td>p←+Zero</td>
<td>p←+Zero</td>
<td>p←+Zero</td>
<td>p←+dQNaN</td>
</tr>
<tr>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
</tr>
<tr>
<td>+Zero</td>
<td>+NZF</td>
<td>+Infinity</td>
<td>+Infinity</td>
<td>+Infinity</td>
<td>+Infinity</td>
</tr>
<tr>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
<td>p←+dQNaN vximz_flag←1</td>
</tr>
<tr>
<td>p←src1</td>
<td>p←src1</td>
<td>p←src1</td>
<td>p←src1</td>
<td>p←src1</td>
<td>p←src1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>QNaN</td>
<td>+Infinity</td>
<td>+Infinity</td>
<td>+Infinity</td>
<td>+Infinity</td>
</tr>
<tr>
<td>p←src1 vxznan_flag←1</td>
<td>p←Q(src1) vxznan_flag←1</td>
<td>p←Q(src1) vxznan_flag←1</td>
<td>p←Q(src1) vxznan_flag←1</td>
<td>p←Q(src1) vxznan_flag←1</td>
<td>p←Q(src1) vxznan_flag←1</td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src2**: For `xsmaddasp`, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3**: For `xsmaddmsp`, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN `0x7FF8_0000_0000_0000`.
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of `x`.
- **A(x,y)**: Return the normalized sum of floating-point value `x` and floating-point value `y`, having unbounded range and precision.
  - **Note**: If `x = -y`, `v` is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)**: Return the normalized product of floating-point value `x` and floating-point value `y`, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
VSX Scalar Maximum Double-Precision

**XX3-form**

### xsmaxdp XT,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>160</td>
<td>256</td>
</tr>
</tbody>
</table>

- \( X_T \leftarrow TX \ || \ T \)
- \( X_A \leftarrow AX \ || \ A \)
- \( X_B \leftarrow BX \ || \ B \)
- reset_xflags()
- src1 \( \leftarrow \) VSR[XA][0:63]
- src2 \( \leftarrow \) VSR[XB][0:63]
- result[0:63] \( \leftarrow \) MaximumDP(src1,src2)
- if(vxsnan_flag) then SetFX(VXSNAN)
- vex_flag \( \leftarrow \) VE \& vxsnan_flag

if( -vex_flag ) then do
  VSR[XT] \( \leftarrow \) result || 0xUUUU_UUUU_UUUU_UUUU
end

Let \( X_T \) be the value \( TX \) concatenated with \( T \).
Let \( X_A \) be the value \( AX \) concatenated with \( A \).
Let \( X_B \) be the value \( BX \) concatenated with \( B \).

Let \( src_1 \) be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let \( src_2 \) be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If \( src_1 \) is greater than \( src_2 \), \( src_1 \) is placed into doubleword element 0 of VSR[XT]. Otherwise, \( src_2 \) is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

The maximum of +0 and −0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 64.

**Special Registers Altered**

- FX VXSNAN
<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

Explanation:

src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2 The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF Nonzero finite number.
Q(x) Return a QNaN with the payload of x.
M(x,y) Return the greater of floating-point value x and floating-point value y.
T(x) The value x is placed in doubleword element 0 of VSR[XT] in double-precision format.
The contents of doubleword element 1 of VSR[XT] are undefined.
FPFR, FR and FI are not modified.
fx(x) If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR\text{VXSNAN}. If VE=1, update of VSR[XT] is suppressed.

Table 64. Actions for xsmaxdp
VSX Scalar Minimum Double-Precision
**XX3-form**

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element 0 of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 65.

**Special Registers Altered**
- FX  VXSNAN
**Table 65. Actions for xvmindp**

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src2)</td>
<td>T(M(xsrc1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(xsrc1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XT].
- **NZF**: Nonzero finite number.
- **Q(x)**: Return a QNaN with the payload of x.
- **M(x,y)**: Return the lesser of floating-point value x and floating-point value y.
- **T(x)**: The value x is placed in doubleword element i (i=0,1) of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
- **fx(x)**: If x is equal to 0, FX is set to 1. x is set to 1.
- **VXSNAN**: Floating-Point Invalid Operation Exception (SNaN) status flag, FPCRVXSNAN. If VE=1, update of VSR[XT] is suppressed.
VSX Scalar Multiply-Subtract Double-Precision XX3-form

**xsmsubadp**

\[
\begin{array}{cccccc}
0 & 6 & T & A & B & 49 & \text{0B}[X]
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 6 & T & A & B & 21 & \text{0B}[X]
\end{array}
\]

**xsmsubmdp**

\[
\begin{array}{cccccc}
0 & 6 & T & A & B & 57 & \text{0B}[X]
\end{array}
\]

- Let XT be the value TX concatenated with T.
- Let XA be the value AX concatenated with A.
- Let XB be the value BX concatenated with B.

For **xsmsubadp**, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 66.

src2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized\(^3\).

See part 2 of Table 66.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

**Special Registers Altered**

FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for `xsmsub(al)m`dp

- **src1** = VSR[XA]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

- **src2** = `xsmsubadp` ? VSR[XT] : VSR[XB]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

- **src3** = `xsmsubadp` ? VSR[XB] : VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

- **tgt** = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>
Table 66. Actions for `xsmsub(a|m)dp`

<table>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th>src3</th>
<th>src2</th>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>p ← -Infinity</td>
<td>p ← dQNaN vximz_flag ← -1</td>
<td>p ← dQNaN vximz_flag ← -1</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Zero</td>
<td>p ← -Zero</td>
<td>p ← -Zero</td>
<td>p ← -Zero</td>
<td>p ← -Zero</td>
<td>p ← dQNaN vximz_flag ← -1</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN vximz_flag ← -1</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← -Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN vximz_flag ← -1</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td>p ← Q(src1) vxsnan_flag ← -1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 2: Subtract</th>
<th>src3</th>
<th>src2</th>
<th>src1</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN vximz_flag ← -1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← -Zero</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← +Zero</td>
<td>v ← Rezd</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN vximz_flag ← -1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
src1: The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2: For `xsmsubadp`, the double-precision floating-point value in doubleword element 0 of VSR[XT].
src3: For `xsmsubadp`, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3: For `xsmsubdmp`, the double-precision floating-point value in doubleword element 0 of VSR[XT].
doQNaN: Default quiet NaN (0x7F8_0000_0000_0000).
NZF: Nonzero finite number.
Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x): Return a QNaN with the payload of x.
S(x,y): Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y): Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p: The intermediate product having unbounded range and precision.
v: The intermediate result having unbounded range and precision.
For \texttt{xsmsubasp}, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For \texttt{xsmsubmsp}, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied\textsuperscript{[1]} by src3, producing a product having unbounded range and precision.

See part 1 of Table 67, “Actions for \texttt{xmsub(a|m)sp}”.

src2 is negated and added\textsuperscript{[2]} to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized\textsuperscript{[3]}.

See part 2 of Table 67, “Actions for \texttt{xmsub(a|m)sp}”.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
Special Registers Altered

FPRF  FR  FI  FX  OX  UX  XX
VXSNAN  VXISI  VXIMZ

VSR Data Layout for xmsub(alm)sp

src1 = VSR[XA]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

src2 = \texttt{xmsubasp} ? VSR[XT] : VSR[XB]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

src3 = \texttt{xmsubasp} ? VSR[XB] : VSR[XT]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>undefined</td>
</tr>
</tbody>
</table>
### Table 67. Actions for xsmsub(a|m)sp

#### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>p ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Zero</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>–Zero</td>
<td>p ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>p ← +Zero</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>p ← –Zero</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← –Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← –Zero</td>
<td>p ← +Zero</td>
<td>p ← –Zero</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td>p ← infinity</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>p ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>p ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
<td>p ← Q(src1)</td>
<td>vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

#### Part 2: Subtract

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>v ← dQNaN</td>
<td>vxisl_flag ← 1</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>–NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>–Zero</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← Rezd</td>
<td>v ← –Zero</td>
<td>v ← –src2</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← +Zero</td>
<td>v ← Rezd</td>
<td>v ← –src2</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← –src2</td>
<td>v ← +Zero</td>
<td>v ← Rezd</td>
<td>v ← –src2</td>
<td>v ← –Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>vxisl_flag ← 1</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

**Explanation:**

| src1 | The double-precision floating-point value in doubleword element 0 of VSR[XA]. |
| src2 | For `xsmsubasp`, the double-precision floating-point value in doubleword element 0 of VSR[XT]. |
| src3 | For `xsmsubasp`, the double-precision floating-point value in doubleword element 0 of VSR[XB]. |
| dQNaN | Default quiet NaN (0x7FF8_0000_0000_0000). |
| NZF | Nonzero finite number. |
| Rezd | Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands. |
| Q(x) | Return a QNaN with the payload of x. |
| S(x,y) | Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. |
| M(x,y) | Return the intermediate product having unbounded range and precision. |
| p | The intermediate product having unbounded range and precision. |
| v | The intermediate result having unbounded range and precision. |
**VSX Scalar Multiply Double-Precision XX3-form**

```
xsmuldp XT,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>B</th>
<th>A</th>
<th>31</th>
<th>16</th>
<th>21</th>
<th>48</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←  TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>←  AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←  BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reset_xflags()

src1 ← VSR[XA](0:63)
src2 ← VSR[XB](0:63)
v(0:inf) ← MultiplyFP(src1,src2)
result(0:63) ← RoundToFP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag) then SetFX(VXIMZ)
if(vxisi_flag) then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag)

if(~vex_flag) then do
  VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
  FPRF ← ClassDP(result)
  FR ← inc_flag
  FI ← xx_flag
end
else do
  FR ← 0b0
  FI ← 0b0
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied\(^1\) by src2, producing a product having unbounded range and precision.

The product is normalized\(^2\).

See Table 68.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 68. Actions for xsmuldp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← M(src1,src2)</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN</td>
<td>v ← M(src1,src2)</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← dQNaN</td>
<td>v ← M(src1,src2)</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **M(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
**VSX Scalar Multiply Single-Precision XX3-form**

\[ \text{xsmulsp } \text{XT,XA,XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>16</th>
<th>XA</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{reset_xflags()} \\
\text{src1} \leftarrow \text{VSR}[32\times AX+A].\text{doubleword}[0] \\
\text{src2} \leftarrow \text{VSR}[32\times BX+B].\text{doubleword}[0] \\
\text{v} \leftarrow \text{MultiplyDP}(\text{src1}, \text{src2}) \\
\text{result} \leftarrow \text{RoundToSP}(\text{RN}, \text{v}) \\
\text{if}(\text{vxsnan_flag}) \text{ then } \text{SetFX}(\text{VXSNAN}) \\
\text{if}(\text{vximz_flag}) \text{ then } \text{SetFX}(\text{VXIMZ}) \\
\text{if}(\text{ox_flag}) \text{ then } \text{SetFX}(\text{OX}) \\
\text{if}(\text{ux_flag}) \text{ then } \text{SetFX}(\text{UX}) \\
\text{if}(\text{xx_flag}) \text{ then } \text{SetFX}(\text{XX}) \\
\text{vex_flag} \leftarrow \text{VE} \& (\text{vxsnan_flag} \mid \text{vximz_flag}) \\
\text{if}(\neg \text{vex_flag}) \text{ then do} \\
\text{VSR}[32\times TX+T].\text{doubleword}[0] \leftarrow \text{ConvertSPtoSP64}(\text{result}) \\
\text{VSR}[32\times TX+T].\text{doubleword}[1] \leftarrow \text{0xUUUU_UUUU_UUUU_UUUU} \\
\text{FPFR} \leftarrow \text{ClassSP}(\text{result}) \\
\text{FR} \leftarrow \text{inc_flag} \\
\text{FI} \leftarrow \text{xx_flag} \\
\text{end} \\
\text{else do} \\
\text{FR} \leftarrow \text{0b0} \\
\text{FI} \leftarrow \text{0b0} \\
\text{end} \\
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied\(^1\) by src2, producing a product having unbounded range and precision.

The product is normalized\(^2\).

See Table 69, “Actions for xsmulsp,” on page 448.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 69. Actions for xsmulsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxisnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← M(src1,src2)</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← M(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxisnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxisnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxisnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← M(src1,src2)</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← M(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxisnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
<td>v ← Q(src1) vxisnan_flag ← 1</td>
</tr>
</tbody>
</table>

### Explanation:

- **src1**: The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2**: The double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of x.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.
**VSX Scalar Negative Absolute Value**

**Double-Precision XX2-form**

\[ \text{xsnabsdp} \quad \text{XT, XB} \]

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>361</th>
<th>SX =</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

\[ \begin{align*}
\text{XT} & \leftarrow \text{TX} || T \\
\text{XB} & \leftarrow \text{BX} || B \\
\text{result(0:63)} & \leftarrow 0b1 || \text{VSR[XB](1:63)} \\
\text{VSR[XT]} & \leftarrow \text{result} || 0xUUUU_UUUU_UUUU_UUUU
\end{align*} \]

Let XT be the value TX concatenated with T. Let XB be the value BX concatenated with B.

The contents of doubleword element 0 of VSR[XB], with bit 0 set to 1, is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered**

None

**VSR Data Layout for xsnabsdp**

\[
\begin{array}{|c|c|}
\hline
\text{src} & \text{VSR[XB]} \\
\hline
\text{DP} & \text{unused} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{tgt} & \text{VSR[XT]} \\
\hline
\text{DP} & \text{undefined} \\
\hline
\end{array}
\]

---

**VSX Scalar Negate Double-Precision**

**XX2-form**

\[ \text{xsnegdp} \quad \text{XT, XB} \]

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>377</th>
<th>SX =</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

\[ \begin{align*}
\text{XT} & \leftarrow \text{TX} || T \\
\text{XB} & \leftarrow \text{BX} || B \\
\text{result(0:63)} & \leftarrow \neg\text{VSR[XB](0)} || \text{VSR[XB](1:63)} \\
\text{VSR[XT]} & \leftarrow \text{result} || 0xUUUU_UUUU_UUUU_UUUU
\end{align*} \]

Let XT be the value TX concatenated with T. Let XB be the value BX concatenated with B.

The contents of doubleword element 0 of VSR[XB], with bit 0 complemented, is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

**Special Registers Altered**

None

**VSR Data Layout for xsnegdp**

\[
\begin{array}{|c|c|}
\hline
\text{src} & \text{VSR[XB]} \\
\hline
\text{DP} & \text{unused} \\
\hline
\text{tgt} & \text{VSR[XT]} \\
\hline
\text{DP} & \text{undefined} \\
\hline
\end{array}
\]
**VSX Scalar Negative Multiply-Add Double-Precision XX3-form**

### xsnmaddadp

<table>
<thead>
<tr>
<th>XT</th>
<th>XA</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

### xsnmaddmdp

<table>
<thead>
<tr>
<th>XT</th>
<th>XA</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

*Let XT be the value TX concatenated with T.*

*Let XA be the value AX concatenated with A.*

*Let XB be the value BX concatenated with B.

**reset_xflags()**

- `src1` ← VSR[XA](0:63)
- `src2` ← “xsnmaddadp” ? VSR[XT](0:63) : VSR[XB](0:63)
- `src3` ← “xsnmaddadp” ? VSR[XB](0:63) : VSR[XT](0:63)
- `v[0:inf]` ← MultiplyAddDP(src1,src3,src2)
- `v[0:inf]` ← VSR[XT](0:63)
- `FR` ← ClassDP(result)
- `FI` ← inc_flag
- `vex_flag` ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

- If `~vex_flag` then do
  - `VSR[XT]` ← result || 0xUUUU_UUUU_UUUU_UUUU
  - `FPRF` ← ClassDP(result)
  - `FR` ← inc_flag
  - `FI` ← xx_flag
- Else do
  - `FR` ← 0
  - `FI` ← 0
- End

For **xsnmaddadp**, do the following.

- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XT].

`src1` is multiplied\(^1\) by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 70.

`src2` is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 70.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 0 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 71, “Scalar Floating-Point Final Result with Negation,” on page 453.

### Special Registers Altered

- FPRF
- FR
- FI
- FX
- OX
- UX
- XX
- VXSNAN
- VXISI
- VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significant accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for `xsnmadd(alm)dp`

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th></th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
<td></td>
<td>undefined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 70. Actions for xsnmadd(aim)dp

#### Part 1: Multiply

<table>
<thead>
<tr>
<th>src3</th>
<th>$-\infty$</th>
<th>$-NZF$</th>
<th>$-Zero$</th>
<th>$+Zero$</th>
<th>$+NZF$</th>
<th>$+Infinity$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow d\text{NaN}$</td>
<td>$p \leftarrow d\text{NaN}$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow -\infty$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
</tr>
<tr>
<td>$-NZF$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow M(\text{src}1,\text{src}3)$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow M(\text{src}1,\text{src}3)$</td>
<td>$p \leftarrow +\infty$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
</tr>
<tr>
<td>$-Zero$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow -\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
</tr>
<tr>
<td>$+Zero$</td>
<td>$p \leftarrow -\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow -\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow +\text{Zero}$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
</tr>
<tr>
<td>$+NZF$</td>
<td>$p \leftarrow +\text{Infinity}$</td>
<td>$p \leftarrow +\text{Infinity}$</td>
<td>$p \leftarrow +\text{Infinity}$</td>
<td>$p \leftarrow +\text{Infinity}$</td>
<td>$p \leftarrow +\text{Infinity}$</td>
<td>$p \leftarrow +\text{Infinity}$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
<td>$p \leftarrow Q(\text{src}3)$</td>
</tr>
<tr>
<td>$+Infinity$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
<td>$p \leftarrow src1$</td>
</tr>
<tr>
<td>QNaN</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
</tr>
<tr>
<td>SNaN</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
<td>$p \leftarrow Q(\text{src}1)$</td>
</tr>
</tbody>
</table>

#### Part 2: Add

<table>
<thead>
<tr>
<th>src2</th>
<th>$-\infty$</th>
<th>$-NZF$</th>
<th>$-Zero$</th>
<th>$+Zero$</th>
<th>$+NZF$</th>
<th>$+Infinity$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$v \leftarrow -\infty$</td>
<td>$v \leftarrow -\infty$</td>
<td>$v \leftarrow -\infty$</td>
<td>$v \leftarrow +\infty$</td>
<td>$v \leftarrow +\infty$</td>
<td>$v \leftarrow +\infty$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
</tr>
<tr>
<td>$-NZF$</td>
<td>$v \leftarrow +\text{Zero}$</td>
<td>$v \leftarrow +\text{Zero}$</td>
<td>$v \leftarrow +\text{Zero}$</td>
<td>$v \leftarrow +\text{Zero}$</td>
<td>$v \leftarrow +\text{Zero}$</td>
<td>$v \leftarrow +\text{Zero}$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
</tr>
<tr>
<td>$-Zero$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
</tr>
<tr>
<td>$+Zero$</td>
<td>$v \leftarrow +\text{Infinity}$</td>
<td>$v \leftarrow +\text{Infinity}$</td>
<td>$v \leftarrow +\text{Infinity}$</td>
<td>$v \leftarrow +\text{Infinity}$</td>
<td>$v \leftarrow +\text{Infinity}$</td>
<td>$v \leftarrow +\text{Infinity}$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
</tr>
<tr>
<td>$+NZF$</td>
<td>$v \leftarrow A(p,\text{src}2)$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow A(p,\text{src}2)$</td>
<td>$v \leftarrow A(p,\text{src}2)$</td>
<td>$v \leftarrow A(p,\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
</tr>
<tr>
<td>$+Infinity$</td>
<td>$v \leftarrow src3$</td>
<td>$v \leftarrow src3$</td>
<td>$v \leftarrow src3$</td>
<td>$v \leftarrow src3$</td>
<td>$v \leftarrow src3$</td>
<td>$v \leftarrow src3$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
<td>$v \leftarrow Q(\text{src}2)$</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow p$</td>
<td>$v \leftarrow src2$</td>
<td>$v \leftarrow src2$</td>
</tr>
</tbody>
</table>

### Explanation:

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** For *xsnmaddap*, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3** For *xsnmaddap*, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)** Return a QNaN with the payload of x.
- **A(x,y)** Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
- **M(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p** The intermediate product having unbounded range and precision.
- **v** The intermediate result having unbounded range and precision.
Case

Special

Normal

Overflow

Returned Results and Status Setting

\[\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{VE} & \text{OE} & \text{UE} & \text{ZE} & \text{XE} & \text{vxsnan\_flag} & \text{vximz\_flag} & \text{vxisi\_flag} & \text{Is } r \text{ inexact?} & \text{Is } r \text{ incremented?} & \text{Is } q \text{ inexact?} & \text{Is } q \text{ incremented?} \\
\hline
- & - & - & - & 0 & 0 & 0 & 0 & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0} \\
0 & - & - & - & - & 0 & 0 & 0 & 0 & - & - & - & - & \text{T(\cdot), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0, fx(VXISI)} \\
0 & - & - & - & 0 & 1 & 0 & 1 & - & - & - & - & \text{T(\cdot), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0, fx(VXIMZ)} \\
0 & - & - & - & 1 & 0 & 0 & 0 & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0, fx(VXSNAN)} \\
0 & - & - & - & 1 & 1 & 0 & 0 & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0, fx(VXSNAN), fx(VXIMZ)} \\
1 & - & - & - & - & 1 & 0 & 1 & - & - & - & - & \text{fx(VXISI), error()} \\
1 & - & - & 0 & 1 & - & - & - & - & - & - & - & \text{fx(VXIMZ), error()} \\
1 & - & - & - & 1 & 0 & 0 & 0 & - & - & - & - & \text{fx(VXSNAN), error()} \\
1 & - & - & - & - & 1 & 1 & 0 & - & - & - & - & \text{fx(VXSNAN), fx(VXIMZ), error()} \\
\hline
- & - & - & - & - & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0} \\
- & - & - & - & 0 & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_0, fx(XX)} \\
- & - & - & - & 0 & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_1, fx(XX)} \\
- & - & - & - & 1 & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_0, fx(XX), error()} \\
- & - & - & - & 1 & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_1, fx(XX), error()} \\
- & 0 & - & 0 & - & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_0, Fr\_0, \beta(\cdot)} \\
- & 0 & - & 0 & - & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_0, \beta(\cdot), fx(OX), fx(XX)} \\
- & 1 & - & 1 & - & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_0, \beta(\cdot), fx(OX), fx(XX), error()} \\
- & 1 & - & 1 & - & - & - & - & - & - & - & - & \text{T(N(\cdot)), FPFF\_ClassFP(\cdot), Fl\_1, Fr\_1, xx(\cdot), error()} \\
\hline
\end{array}\]

Explanation:

- The results do not depend on this condition.
- ClassFP(x) Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
- fx(x) FX is set to 1 if x=0. x is set to 1.
- \(\beta\) Wrap adjust, where \(\beta = 2^{1536}\) for double-precision and \(\beta = 2^{192}\) for single-precision.
- q The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range.
- r The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range.
- v The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- FI Floating-Point Fraction Inexact status flag, FPSCR_FI. This status flag is nonsticky.
- FR Floating-Point Fraction Rounded status flag, FPSCR_FR.
- OX Floating-Point Overflow Exception status flag, FPSCR_OX.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- N(x) The value x is is negated by complementing the sign bit of x.
- T(x) The value x is placed in element 0 of VSR[XT] in the target precision format.
- UX Floating-Point Underflow Exception status flag, FPSCR_UX.
- VXSNAN Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR_VXSNAN.
- VXIMZ Floating-Point Invalid Operation Exception (Infinity \times Zero) status flag, FPSCR_VXIMZ.
- VXISI Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCR_VXISI.
- XX Floating-Point Inexact Exception status flag, FPSCR_XX. The flag is a sticky version of FPSCR_FI. When FPSCR_FI is set to a new value, the new value of FPSCR_XX is set to the result of ORing the old value of FPSCR_XX with the new value of FPSCR_FI.

Table 71. Scalar Floating-Point Final Result with Negation
Tiny

<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 0, FR ← 0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 1, FR ← 0, fx(UX), bx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 1, FR ← 1, fx(UX), bx(XX)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 1, FR ← 0, fx(UX), bx(X), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 1, FR ← 0, fx(UX), bx(X), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 1, FR ← 0, fx(UX), bx(X), error()</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T(N(0)), FPRF ← ClassFP(N(0)), FI ← 1, FR ← 0, fx(UX), bx(X), error()</td>
</tr>
</tbody>
</table>

**Explanation:**

- The results do not depend on this condition.
- ClassFP(x) Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
- fx(x) FX is set to 1 if x = 0. x is set to 1.
- \( \beta \) Wrap adjust, where \( \beta = 2^{1536} \) for double-precision and \( \beta = 2^{192} \) for single-precision.
- \( q \) The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range.
- \( r \) The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range.
- \( v \) The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- FI Floating-Point Fraction Inexact status flag, FPSCRFI.
- FR Floating-Point Fraction Rounded status flag, FPSCRFR.
- OX Floating-Point Overflow Exception status flag, FPSCRUX.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- N(x) The value x is is negated by complementing the sign bit of x.
- T(x) The value x is placed in element 0 of VSR[XT] in the target precision format.
- ux The contents of the remaining element(s) of VSR[XT] are undefined.
- UX Floating-Point Underflow Exception status flag, FPSCRUX.
- VXSNAN Floating-Point Invalid Operation Exception (SNan) status flag, FPSCRVXSNAN.
- VXIMZ Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCRVXIMZ.
- VXISI Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCRVXISI.
- XX Floating-Point Inexact Exception status flag, FPSCRXX. The flag is a sticky version of FPSCRXX. When FPSCRXX is set to a new value, the new value of FPSCRXX is set to the result of ORing the old value of FPSCRXX with the new value of FPSCRXX.

Table 71.Scalar Floating-Point Final Result with Negation (Continued)
VSX Scalar Negative Multiply-Add
Single-Precision XX3-form

For \texttt{xsnmaddasp}, do the following.
- Let \texttt{src1} be the double-precision floating-point value in doubleword element 0 of \texttt{VSR[XA]}.
- Let \texttt{src2} be the double-precision floating-point value in doubleword element 0 of \texttt{VSR[XT]}.
- Let \texttt{src3} be the double-precision floating-point value in doubleword element 0 of \texttt{VSR[XB]}.

\texttt{src1} is multiplied\textsuperscript{1} by \texttt{src3}, producing a product having unbounded range and precision.

See part 1 of Table 72, “Actions for \texttt{xsnmadd(a|m)sp},” on page 457.

\texttt{src2} is added\textsuperscript{2} to the product, producing a sum having unbounded range and precision.

The sum is normalized\textsuperscript{3}.

See part 2 of Table 72, “Actions for \texttt{xsnmadd(a|m)sp},” on page 457.

The intermediate result is rounded to single-precision using the rounding mode specified by \texttt{RN}.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element 0 of \texttt{VSR[XT]} in double-precision format.

The contents of doubleword element 1 of \texttt{VSR[XT]} are undefined.

\texttt{FPRF} is set to the class and sign of the result as represented in single-precision format. \texttt{FR} is set to indicate if the result was incremented when rounded. \texttt{FI} is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, \texttt{VSR[XT]} and \texttt{FPRF} are not modified, and \texttt{FR} and \texttt{FI} are set to 0.

See Table 71, “Scalar Floating-Point Final Result with Negation,” on page 453.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Special Registers Altered

<table>
<thead>
<tr>
<th>FPRF</th>
<th>FR</th>
<th>FI</th>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXSNAN</td>
<td>VXISI</td>
<td>VXIMZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### VSR Data Layout for xsnmadd(a|m)sp

src1 = VSR[XA]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

csrc2 = \text{\texttt{xsnmadda(dp|sp)}} ? VSR[XT] : VSR[XB]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

csrc3 = \text{\texttt{xsnmadda(dp|sp)}} ? VSR[XB] : VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
</table>

0       64       127
Table 72. Actions for xsnmadd(a|m)sp

<table>
<thead>
<tr>
<th>src1</th>
<th>src3</th>
<th>src2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(x,y)</td>
<td>Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(x,y)</td>
<td>Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q(x)</td>
<td>Return a QNaN with the payload of x.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NZF</td>
<td>Nonzero finite number.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reszd</td>
<td>Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q(x)</td>
<td>Return a QNaN with the payload of x.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(x,y)</td>
<td>Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M(x,y)</td>
<td>Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>The intermediate product having unbounded range and precision.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>The intermediate result having unbounded range and precision.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For \texttt{xsnmsubdmp}, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 73.

src2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 73.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 71, “Scalar Floating-Point Final Result with Negation,” on page 453.

Special Registers Altered
FPRF FR FI FX OX UX XX VXSNAN VXSI VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
**VSR Data Layout for xsnmsub(al)m)dp**

src1 = VSR[XA]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

src2 = `xsnmsubadp` ? VSR[XT] : VSR[XB]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

src3 = `xsnmsubadp` ? VSR[XB] : VSR[XT]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>
### Table 73. Actions for xsnmsub(alim)dp

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
<th>$\text{vm}_{\text{flag}}$</th>
<th>$\text{vxsnan}_{\text{flag}}$</th>
<th>$\text{vximz}_{\text{flag}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>p &amp; $+\text{Infinity}$</td>
<td>p &amp; $-\text{Infinity}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td></td>
</tr>
<tr>
<td>p &amp; $-\text{Infinity}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td></td>
</tr>
<tr>
<td>p &amp; $-\text{Infinity}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td></td>
</tr>
<tr>
<td>p &amp; $-\text{Infinity}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td>p &amp; $\text{QNaN}$</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** For xsnmsubadp, the double-precision floating-point value in doubleword element 0 of VSR[XT].
- **src3** For xsnmsubadp, the double-precision floating-point value in doubleword element 0 of VSR[XB].
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)** Return a QNaN with the payload of x.
- **S(x,y)** Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p** The intermediate product having unbounded range and precision.
- **v** The intermediate result having unbounded range and precision.
VSX Scalar Negative Multiply-Subtract
Single-Precision XX3-form

\text{For } \text{\texttt{xsnmsubasp}}, \text{ do the following.}
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[AX].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[BX].

\text{For } \text{\texttt{xsnmsubmsp}}, \text{ do the following.}
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[AX].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[BX].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

\begin{enumerate}
\item src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.
\item See part 1 of Table 74, “Actions for \texttt{xsnmsub(a|m)sp},” on page 463.
\item src2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.
\item The sum is normalized\(^3\).
\item See part 2 of Table 74, “Actions for \texttt{xsnmsub(a|m)sp},” on page 463.
\item The intermediate result is rounded to single-precision using the rounding mode specified by RN.
\item See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.
\item The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.
\item The contents of doubleword element 1 of VSR[XT] are undefined.
\item FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.
\item If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.
\item See Table 71, “Scalar Floating-Point Final Result with Negation,” on page 453.
\end{enumerate}

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Special Registers Altered

| FPRF | FR  | FI  | FX  | OX  | UX  | XX  | VXSNAN | VXISI | VXIMZ |

### VSR Data Layout for xsnmsub(a|m)sp

**src1 = VSR[XA]**

| DP | unused |

**src2 = xsnmsubasp ? VSR[XT] : VSR[XB]**

| DP | unused |

**src3 = xsnmsubasp ? VSR[XB] : VSR[XT]**

| DP | unused |

**tgt = VSR[XT]**

| DP | undefined |

| 0 | 64 | 127 |
### Table 74. Actions for xsnmsub(a|m)sp

**Part 1: Multiply**

<table>
<thead>
<tr>
<th>src3</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← -Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← -Zero</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← +Zero</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← dQNaN vximz_flag ← 1</td>
<td>p ← +Zero</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← M(src1,src3)</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← -Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← -Infinity</td>
<td>p ← +Infinity</td>
<td>p ← src3</td>
<td>p ← Q(src3) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Part 2: Subtract**

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN vximz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← -Zero</td>
<td>v ← -src2</td>
<td>v ← -src2</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← +src2</td>
<td>v ← Rezd</td>
<td>v ← +src2</td>
<td>v ← +src2</td>
<td>v ← +src2</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(p,src2)</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← S(p,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN &amp; src1 a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1**: The double-precision floating-point value in VSR[XA].doubleword[0].
- **src2**: For **xsnmsubasp**, the double-precision floating-point value in VSR[XT].doubleword[0]. For **xsnmsubmsp**, the double-precision floating-point value in VSR[XB].doubleword[0].
- **src3**: For **xsnmsubasp**, the double-precision floating-point value in VSR[XB].doubleword[0]. For **xsnmsubmsp**, the double-precision floating-point value in VSR[XT].doubleword[0].
- **dQNaN**: Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of x.
- **S(x,y)**: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
VSX Scalar Round to Double-Precision Integer
using round to Nearest Away XX2-form

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in
doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode
Round to Nearest Away.

The result is placed into doubleword element 0 of
VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are
undefined.

FPRF is set to the class and sign of the result. FR is
set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs,
VSR[XT] and FPRF are not modified, and FR and FI
are set to 0.

Special Registers Altered

FPRF  FR=0b0  FI=0b0  FX  VXSNAN

VSR Data Layout for xsrdpi

src = VSR[XB]

tgt = VSR[XT]
VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form

**xsrdpic XT,XB**

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>//</th>
<th>B</th>
<th>107</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

**Special Registers Altered**

| FPRF | FR | FI | FX | XX | VXSNAN |

---

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].
VSX Scalar Round to Double-Precision Integer using round toward -Infinity  XX2-form

**xsrdpim**

<table>
<thead>
<tr>
<th>XT</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x60</td>
<td>T///B</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} \| \text{T} \\
\text{XB} & \leftarrow \text{BX} \| \text{B} \\
\text{reset_xflags()} \\
\text{result}(0:63) & \leftarrow \text{RoundToDPIntegerFloor}(\text{VSR}[\text{XB}](0:63)) \\
\text{if}(\text{vxsnan_flag}) & \text{ then SetFX(\text{VXSNAN})} \\
\text{FR} & \leftarrow 0b0 \\
\text{FI} & \leftarrow 0b0 \\
\text{vex_flag} & \leftarrow \text{VE} \& \text{vxsnan_flag} \\
\text{if}(\text{-vex_flag} \text{ then do} \\
& \hspace{1cm} \text{VSR}[\text{XT}] \leftarrow \text{result} || 0xUUUU_UUUU_UUUU_UUUU \\
& \hspace{1cm} \text{FPRF} \leftarrow \text{ClassDP(result)} \\
\text{end}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

**Special Registers Altered**

FPRF FR=0b0 FI=0b0 FX VXSNAN

**VSR Data Layout for xsrdpim**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 4 127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 4 127</td>
</tr>
</tbody>
</table>

VSX Scalar Round to Double-Precision Integer using round toward +Infinity  XX2-form

**xsrdpip**

<table>
<thead>
<tr>
<th>XT</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x60</td>
<td>T///B</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} \| \text{T} \\
\text{XB} & \leftarrow \text{BX} \| \text{B} \\
\text{reset_xflags()} \\
\text{result}(0:63) & \leftarrow \text{RoundToDPIntegerCeil}(\text{VSR}[\text{XB}](0:63)) \\
\text{if}(\text{vxsnan_flag}) & \text{ then SetFX(\text{VXSNAN})} \\
\text{FR} & \leftarrow 0b0 \\
\text{FI} & \leftarrow 0b0 \\
\text{vex_flag} & \leftarrow \text{VE} \& \text{vxsnan_flag} \\
\text{if}(\text{-vex_flag} \text{ then do} \\
& \hspace{1cm} \text{VSR}[\text{XT}] \leftarrow \text{result} || 0xUUUU_UUUU_UUUU_UUUU \\
& \hspace{1cm} \text{FPRF} \leftarrow \text{ClassDP(result)} \\
\text{end}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

**Special Registers Altered**

FPRF FR=0b0 FI=0b0 FX VXSNAN

**VSR Data Layout for xsrdpip**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 4 127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 4 127</td>
</tr>
</tbody>
</table>
VSX Scalar Round to Double-Precision Integer using round toward Zero \textit{XX2-form}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
0 & 60 & T & /// & B & 89 & XN \\
\hline
\end{tabular}

\begin{align*}
\text{XT} & \leftarrow TX || T \\
\text{XB} & \leftarrow BX || B \\
\text{result} & \leftarrow \text{RoundToDPIntegerTrunc}(	ext{VSR[XB]} | 0:63) \\
\text{if(vxsnan}\_\text{flag}) & \text{then SetFX(VXSNAN)} \\
\text{FR} & \leftarrow 0b0 \\
\text{FI} & \leftarrow 0b0 \\
\text{vex}\_\text{flag} & \leftarrow \text{VE} \& \text{vxsnan}\_\text{flag} \\
\text{if}(-\text{vex}\_\text{flag}) & \text{then do} \\
& \text{VSR[XT]} \leftarrow \text{result} || 0xUUUU\_UUUU\_UUUU\_UUUU \\
& \text{FPRF} \leftarrow \text{ClassDP(result)} \\
\text{end}
\end{align*}

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

\textbf{Special Registers Altered}

\begin{tabular}{lllll}
\text{FPRF} & \text{FR=0b0} & \text{FI=0b0} & \text{FX} & \text{VXSNAN}
\end{tabular}

\textbf{VSR Data Layout for xsrdpiz}

\begin{tabular}{|c|}
\hline
src = VSR[XB] \\
\hline
\end{tabular}

\begin{tabular}{|c|}
\hline
DP & unused \\
\hline
\end{tabular}

\begin{tabular}{|c|}
\hline
tgt = VSR[XT] \\
\hline
\end{tabular}

\begin{tabular}{|c|}
\hline
DP & undefined \\
\hline
\end{tabular}
VSX Scalar Reciprocal Estimate
Double-Precision XX2-form

xsredp XT, XB

<table>
<thead>
<tr>
<th>60</th>
<th>127</th>
<th>90</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←</td>
<td>TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reset_xflags()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v(0:inf) ← ReciprocalEstimateDP(VSR[XB][0:63])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result(0:63) ← RoundToDP(RN,v)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(vxsnan_flag) then SetFX(VXSNAN)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(ox_flag) then SetFX(OX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(ux_flag) then SetFX(UX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(zx_flag) then SetFX(ZX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vex_flag ← VE &amp; vxsnan_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zex_flag ← ZE &amp; zx_flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if( ~vex_flag &amp; ~zex_flag ) then do</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSR[XT] ← result</td>
<td></td>
<td>0xUUUU_UUUU_UUUU_UUUU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPRF ← ClassDP(result)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR ← 0bU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI ← 0bU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T. Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src.

That is,

\[
\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>-Zero</td>
<td>None</td>
</tr>
<tr>
<td>-Zero</td>
<td>-Infinity</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if ZE=1.
2. No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered
FPRF FR=0bU FI=0bU FX OX UX XX=0bU VXSNAN

VSR Data Layout for xsredp
src = VSR[XB]

```
+------------------+
|                  |
|      DP          |
|      unused      |
+------------------+

tgt = VSR[XT]

```

```
+------------------+
|                  |
|      DP          |
|      undefined   |
+------------------+

0  64  127

468 Power ISA™ - Book I
VSX Scalar Reciprocal Estimate
Single-Precision XX2-form

xsresp \( \rightarrow \) XT, XB

| 60 | 60 T 11 || 16 B 21 | 26 | B1 |

reset_xflags()

src ← VSR[32×BX+B].doubleword[0]
v ← ReciprocalEstimateDP(src)
result ← RoundToSP[RN,v]

if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(zx_flag) then SetFX(ZX)
if(0bU) then SetFX(XX)

vex_flag ← VE & vxsnan_flag
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do

VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
FPRF ← ClassSP(result)
FR ← 0bU
FI ← 0bU
end
else do
FR ← 0b0
FI ← 0b0
end

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

Let \( src \) be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of \( src \) is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of \( src \) would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate has a relative error in precision no greater than one part in \( 16384 \) of the reciprocal of \( src \). That is,

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>−Infinity</td>
<td>−Zero</td>
<td>None</td>
</tr>
<tr>
<td>−Zero</td>
<td>−Infinity(^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity(^1)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^2)</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**

FPRF FR=0bU FI=0bU FX=0bU OX UX ZX=0bU VXSNAN

**VSR Data Layout for xsresp**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

---

Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 469
VSX Scalar Round to Single-Precision
XX2-form

xsrsp  XT,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>281</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reset_xflags()

src ← VSR[12×BX+B].doubleword[0]
result ← RoundToSP(RN,src)

if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)

vex_flag ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[12×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[12×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR ← inc_flag
   FI ← xx_flag
end
else do
   FR ← 0b0
   FI ← 0b0
end

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in
doubleword element 0 of VSR[XB].

src is rounded to single-precision using the rounding
mode specified by the Floating-Point Rounding Control
field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result
Handling,” on page 402.

The result is placed into doubleword element 0 of
VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are
undefined.

FPRF is set to the class and sign of the result as
represented in single-precision format.

If a trap-enabled invalid operation exception occurs,
VSR[XT] and FPRF are not modified.

Special Registers Altered
   FPR FR FI FX OX UX XX VXSNAN
VSX Scalar Reciprocal Square Root Estimate
Double-Precision XX2-form

**xsrsqrtedp XT,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>B</th>
<th>/ / /</th>
<th>16</th>
<th>21</th>
<th>74</th>
<th>BXX</th>
</tr>
</thead>
</table>

**xt** ← TX || T

**xb** ← BX || B

reset_xflags()

v(0:inf) ← ReciprocalSquareRootEstimateDP(VSR[XB](0:63))

result(0:63) ← RoundToDP(FR, v)

if(vxsnan_flag) then SetFX(VXSNAN)

if(vxsqrt_flag) then SetFX(VXSQRT)

if(zx_flag) then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)

zex_flag ← ZE & zx_flag

if(~vex_flag & ~zx_flag) then do

VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU

FPRF ← ClassDP(result)

FR ← 0bU

FI ← 0bU

end

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

\[
\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>QNaN1</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Finite</td>
<td>QNaN1</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Zero</td>
<td>–Infinity2</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity2</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN1</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**

FPRF FR=0bU FI=0bU FX OX UX
XX=0bU VXSNAN VXSQRT

**VSR Data Layout for xsrsqrtedp**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>
VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form

**xsrsqrtesp XT,XB**

```
<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>10</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>/ /</td>
<td>16</td>
<td>21</td>
<td>28</td>
</tr>
</tbody>
</table>
```

reset_xflags()  
```
src ← VSR[12×BX+B].doubleword[0]  
v ← ReciprocalSquareRootEstimateDP(src)  
result ← RoundToSP(RN, v)  
if(vxsnan_flag) then SetFX(VXSNAN)  
if(vxsqrt_flag) then SetFX(VXSQRT)  
if(0bU) then SetFX(XX)  
if(zx_flag) then SetFX(ZX)  
vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)  
zex_flag ← ZE & zx_flag  
if(-vex_flag & -zex_flag) then do  
  VSR[12×TX+T].doubleword[0] ← ConvertSPtoSP64(result)  
  VSR[12×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU  
  FPRF ← ClassSP(result)  
  FR ← 0b0  
  FI ← 0b0  
end  
else do  
  FR ← 0b0  
  FI ← 0b0  
end
```

Let XT be the value TX concatenated with T.  
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

\[
\frac{\text{estimate} - \frac{1}{\sqrt{src}}}{\frac{1}{\sqrt{src}}} \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

```
<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Finite</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Zero</td>
<td>–Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^1)</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>
```

1. No result if VE=1.  
2. No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**

| FPRF FR=0bU FI=0bU FX ZX XX=0bU VXSNAN | VXSQRT |

**VSR Data Layout for xsrsqrtesp**

```
src = VSR[XB]  
tgt = VSR[XT]
```

```
<table>
<thead>
<tr>
<th>DP</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>DP</th>
<th>undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>
```

472  
Power ISA™ - Book I
VSX Scalar Square Root Double-Precision XX2-form

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, "Scalar Floating-Point Final Result," on page 403.

### Special Registers Altered

- FPRF
- FR
- FI
- FX
- XX
- VXSNAN
- VXSQRT

### VSR Data Layout for xssqrtdp

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>unused</td>
</tr>
<tr>
<td>DP</td>
<td>undefined</td>
</tr>
</tbody>
</table>

#### Table 75: Actions for xssqrtdp

<table>
<thead>
<tr>
<th>src</th>
<th>v ← dQNaN</th>
<th>v ← dQNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
</tr>
<tr>
<td>← SQRT(src)</td>
<td>v ← +Infinity</td>
<td>v ← src</td>
</tr>
<tr>
<td>vxsqrt_flag ← 1</td>
<td>vxsqrt_flag ← 1</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- src The double-precision floating-point value in doubleword element 0 of VSR[XB].
- dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF Nonzero finite number.
- SQRT(x) The unbounded-precision square root of the floating-point value x.
- Q(x) Return a QNaN with the payload of x.
- v The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Square Root Single-Precision
XX-form

xssqrtsp XT,XB

reset_xflags();

src ← VSR[12×BX+B].doubleword[0]
v ← SquareRootDP(src)
result ← RoundToSP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)

if(-vex_flag ) then do
    VSR[12×TX+T].doubleword[0] ← ConvertToDP(result)
    VSR[12×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
    FPRF ← ClassSP(result)
    FR ← inc_flag
    FI ← xx_flag
end
else do
    FR ← 0b0
    FI ← 0b0
end

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 75.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered
FPRF FR FI FX OX UX XX
VXSNAN VXSQRT

VSR Data Layout for xssqrtsp
src = VSR[XB]
tgt = VSR[XT]

<table>
<thead>
<tr>
<th>src</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v←dQNaN</td>
<td>v←dQNaN</td>
<td>v←+Zero</td>
<td>v←+Zero</td>
<td>v←SQRT(src)</td>
<td>v←+Infinity</td>
<td>v←src</td>
<td>v←Q(src)</td>
<td>vxsnan_flag←1</td>
</tr>
</tbody>
</table>

Explanation:
src The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
NZF Nonzero finite number.
SQRT(x) The unbounded-precision and exponent range square root of the floating-point value x.
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.

Table 76 Actions for xssqrtsp
**VSX Scalar Subtract Double-Precision XX3-form**

Let XT be the value TX concatenated with T. Let XA be the value AX concatenated with A. Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added\(^1\) to src1, producing a sum having unbounded range and precision.

See Table 77.

The sum is normalized\(^2\).

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
Table 77. Actions for xssubdp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← -Infinity</td>
<td>v ← +NZF</td>
<td>v ← -Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1, src2)</td>
<td>v ← src1</td>
<td>v ← +NZF</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

Explanation:

src1 The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2 The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
NZF Nonzero finite number.
Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y) Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 477

**VSX Scalar Subtract Single-Precision XX3-form**

xssubsp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>8</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>16</td>
<td>21</td>
<td>30</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

reset_xflags()

src1 ← VSR[32×XX+AX].doubleword[0]
src2 ← VSR[32×XX+BX].doubleword[0]
v ← AddDP[(src1,MegateDP(src2))]
result ← RoundToSP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag) then SetFX(VXISI)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxisi_flag)

if (-vex_flag) then do
  VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
  FPRF ← ClassSP(result)
  FR ← inc_flag
  FI ← xx_flag
  end
else do
  FR ← 0b0
  FI ← 0b0
  end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added[1] to src1, producing the sum, v, having unbounded range and precision.

See Table 78, “Actions for xssubsp,” on page 478.

v is normalized[2] and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 78. Actions for xssubsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN vxsnan_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element 0 of VSR[XA].
- **src2** The double-precision floating-point value in doubleword element 0 of VSR[ XB].
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- **S(x,y)** Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Scalar Test for software Divide
Double-Precision XX3-form

**xstdivdp** BF,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>//</th>
<th>A</th>
<th>B</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

XA ← AX || A
XB ← BX || B
src1 ← VSR[XA][0:62]
src2 ← VSR[XB][0:63]
e_a ← VSR[XA][1:11] - 1023
e_b ← VSR[XB][1:11] - 1023
fe_flag ← IsnNaN(src1) | IsnInf(src1) |
IsnNaN(src2) | IsnInf(src2) | IsnZero(src2) |
( e_b <= -1022 ) |
( e_b >= 1021 ) |
( !IsnZero(src1) && ( e_a - e_b ) >= 1023 ) |
( !IsnZero(src1) && ( e_a - e_b ) <= -1021 ) |
( !IsnZero(src1) && ( e_a - e_b ) <= -970 ) |
fg_flag ← IsnInf(src1) | IsnInf(src2) |
IsnZero(src2) | IsnDen(src2)
fl_flag ← xsredp_error() <= 2^-14
CR[BF] ← 0b1 || fg_flag || fe_flag || 0b0

Let XA be the value AX concatenated with A.

Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let e_a be the unbiased exponent of src1.

Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.
- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- src1 is not a zero and e_a is less than or equal to -970

Otherwise fe_flag is set to 0.

fg_flag is set to 1 for any of the following conditions.
- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

Otherwise fg_flag is set to 0.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.
**VSX Scalar Test for software Square Root Double-Precision XX2-form**

```
<table>
<thead>
<tr>
<th>xstsqrtdp</th>
<th>BF,XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>BF</td>
</tr>
<tr>
<td>60</td>
<td>//</td>
</tr>
<tr>
<td>60</td>
<td>///</td>
</tr>
<tr>
<td>106</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>106</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>106</td>
<td>B</td>
</tr>
</tbody>
</table>

Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let e_b be the unbiased exponent of src.

- fe_flag is set to 1 for any of the following conditions.
  - src is a zero, a NaN, an infinity, or a negative value.
  - e_b is less than or equal to -970

Otherwise fe_flag is set to 0.

- fg_flag is set to 1 for any of the following conditions.
  - src is a zero, an infinity, or a denormalized value.

Otherwise fg_flag is set to 0.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered
CR[BF]

**VSX Vector Absolute Value Double-Precision XX2-form**

```
<table>
<thead>
<tr>
<th>xvabsdp</th>
<th>XT,XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
</tr>
<tr>
<td>60</td>
<td>///</td>
</tr>
<tr>
<td>60</td>
<td>///</td>
</tr>
<tr>
<td>473</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>473</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>473</td>
<td>B</td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.

Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 set to 0, is placed into doubleword element i of VSR[XT].

Special Registers Altered
None

**VSR Data Layout for xvabsdp**

```
<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>unused</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>
```

**VSR Data Layout for xstsqrtdp**

```
<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>unused</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

480  Power ISA™ - Book I
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

VSX Vector Absolute Value Single-Precision

**XX2-form**

xvabssp XT, XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>11</th>
<th>B</th>
<th>21</th>
<th>409</th>
<th>31</th>
</tr>
</thead>
</table>

XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← 0b0 || VSR[XB]{i+1:i+31}
end

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 set to 0, is placed into word element i of VSR[XT].

**Special Registers Altered**
None

VSR Data Layout for xvabssp

src = VSR[XB]

```
  SP  SP  SP  SP
```

tgt = VSR[XT]

```
  SP  SP  SP  SP
```

0 32 64 96 127
VSX Vector Add Double-Precision XX3-form

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvadddp

src1 = VSR[XA]

src2 = VSR[XB]

tgt = VSR[XT]

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>vxsi_flag ← 1</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Rezd</td>
<td>v ← -Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>vxsi_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsi_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsi_flag ← 1</td>
<td>v ← Q(src1)</td>
<td>vxsi_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:

src1 The double-precision floating-point value in doubleword element i of VSR[XA] (where i \( \in \{0,1\} \)).
src2 The double-precision floating-point value in doubleword element i of VSR[XB] (where i \( \in \{0,1\} \)).
dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
NZF Nonzero finite number.
Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y) Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.

Table 79. Actions for xvadddp (element i)
### Returned Results and Status Setting

| Case | VE | OE | UE | ZE | XE | vxsnan_flag | vximz_flag | vxidi_flag | vxzdz_flag | vxsqrt_flag | zx_flag | Is r inexact? (r ≥ v) | Is q incremented? (|q| > |v|) |
|------|----|----|----|----|----|-------------|------------|------------|------------|-------------|--------|---------------------|---------------------|
| Special | | | | | | | | |
| | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | - | - | - | T(r) |
| | - | 0 | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(ZX) |
| | 0 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXSQRT) |
| | 0 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXZDZ) |
| | 0 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXIMZ) |
| | 0 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXISI) |
| | 0 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXSNAN) |
| | 0 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXSNAN), fx(VXIMZ) |
| | 1 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXSQRT) |
| | 1 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXZDZ), error() |
| | 1 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXIDI), error() |
| | 1 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXISI), error() |
| | 1 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXIMZ), error() |
| | 1 | - | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(VXSNAN), fx(VXIMZ), error() |
| Normal | | | | | | | | |
| | - | 0 | - | - | - | - | - | - | 0 | - | - | - | T(r) |
| | - | 0 | - | - | - | - | - | - | 0 | - | - | - | T(r), fx(XX) |
| | - | 0 | - | - | - | - | - | - | 0 | - | - | - | T(r), fx(XX) |
| | - | 0 | - | - | - | - | - | - | 0 | - | - | - | T(r), fx(XX), error() |
| | - | 0 | - | - | - | - | - | - | 0 | - | - | - | T(r), fx(XX), error() |
| | - | 1 | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(XX), error() |
| | - | 1 | - | - | - | - | - | - | 1 | - | - | - | T(r), fx(XX), error() |
| Explanation: |
| – | The results do not depend on this condition. |
| fx(x) | FX is set to 1 if x≠0, x is set to 1. |
| q | The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range. |
| r | The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range. |
| v | The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range. |
| OX | Floating-Point Overflow Exception status flag, FPSCR_OX. |
| error() | The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements. |
| T(x) | The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4} for results with 32-bit elements). |
| UX | Floating-Point Underflow Exception status flag, FPSCR_UX. |
| VXSNAN | Floating-Point Invalid Operation Exception (SNan) status flag, FPSCR_VXSNAN. |
| VXSQRT | Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, FPSCR_VXSQRT. |
| VXIDI | Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, FPSCR_VXIDI. |
| VXIMZ | Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, FPSCR_VXIMZ. |
| VXISI | Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCR_VXISI. |
| VXZDZ | Floating-Point Inexact Exception status flag, FPSCR_VXZDZ. |
| XX | Floating-Point Zero Divide Exception status flag, FPSCR_VXZDZ. |

Table 80. Vector Floating-Point Final Result
### Table 80. Vector Floating-Point Final Result (Continued)

<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>ZE</th>
<th>XZ</th>
<th>VXSnan_flag</th>
<th>VXInsNaN_flag</th>
<th>VXZDZ_flag</th>
<th>VXiI_flag</th>
<th>VXiD_flag</th>
<th>VXiZ_flag</th>
<th>VXdiag_flag</th>
<th>VXtag_flag</th>
<th>ZX</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>T(c), fx(OX), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>0</td>
<td>T(c), fx(OX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>0</td>
<td>T(c), fx(OX), fx(XX), error()</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>0</td>
<td>T(c), fx(OX), fx(XX), error()</td>
</tr>
</tbody>
</table>

**Explanation:**
- The results do not depend on this condition.
- `fx(x)` FX is set to 1 if `x=0`. `x` is set to 1.
- `q` The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range.
- `r` The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range.
- `v` The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- `OX` Floating-Point Overflow Exception status flag, FPSCR_OX.
- `error()` The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
- `T(x)` The value `x` is placed in element `i` of VSR[XT] in the target precision format (where `i \in \{0,1\}` for results with 64-bit elements, and `i \in \{0,1,3,4\}` for results with 32-bit elements).
- `UX` Floating-Point Underflow Exception status flag, FPSCR_UX.
- `VXSnan` Floating-Point Invalid Operation Exception (SNaN) status flag, FPSCR_VXSnan.
- `VXsOrt` Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, FPSCR_VXsOrt.
- `VXidi` Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, FPSCR_VXidi.
- `VXimi` Floating-Point Invalid Operation Exception (Infinity × 0) status flag, FPSCR_VXimi.
- `VXisi` Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, FPSCR_VXisi.
- `VXzdZ` Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, FPSCR_VXzdZ.
- `XX` Floating-Point Inexact Exception status flag, FPSCR_XX. The flag is a sticky version of FPSCR_XX. When FPSCR_XX is set to a new value, the new value of FPSCR_XX is set to the result of ORing the old value of FPSCR_XX with the new value of FPSCR_XX.
- `ZX` Floating-Point Zero Divide Exception status flag, FPSCR_ZX.
VSX Vector Add Single-Precision XX3-form

The VSX Vector Add Single-Precision instruction, `vaddsp XT,XA,XB`, is defined as follows:

### Instruction Details

- **Operation**: Adds single-precision floating-point operands from `VSR[XA]` and `VSR[XB]` and stores the result in `VSR[XT]`.
- **Syntax**: `vaddsp XT,XA,XB`
- **Description**: For each vector element `i` from 0 to 127 by 32:

  - Let `src1` be the single-precision floating-point operand in word element `i` of `VSR[XA]`.
  - Let `src2` be the single-precision floating-point operand in word element `i` of `VSR[XB]`.
  - `src2` is added to `src1`, producing a sum having unbounded range and precision.
  - The sum is normalized.
  - The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

### Functionality

- **Special Registers Altered**:
  - FX, OX, UX, XX, VXSNAN, VXISI

### Data Layout

**VSR Data Layout for vaddsp**

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

**src1 = VSR[XA]**

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

**src2 = VSR[XB]**

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

**tgt = VSR[XT]**

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>

### Notes

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
## Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

### Table 81: Actions for `{vaddsp}` (element i)

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -dQNaN</td>
<td>vxisi_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Infinity</td>
<td>v ← A(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← A(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -dQNaN</td>
<td>vxisi_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1 vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

src1 The single-precision floating-point value in word element i of VSR[XA] (where i ∈ \{0,1,2,3\}).

src2 The single-precision floating-point value in word element i of VSR[XB] (where i ∈ \{0,1,2,3\}).

dQNaN Default quiet NaN (0x7FC0_0000).

NZF Nonzero finite number.

Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).

A(x,y) Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.

Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).

Q(x) Return a QNaN with the payload of x.

v The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Vector Compare Equal To Double-Precision [ & Record ] XX3-form

\[
\text{xvcmpeqdp}. \ XT,XA,XB \ (Rc=0) \\
\text{xvcmpeqdp}. \ XT,XA,XB \ (Rc=1)
\]

\[
\begin{array}{cccccccccc}
60 & 6 & T & A & B & Rc & 99 & VSR[XT] \\
0 & 6 & 11 & 16 & 21 & 22 & 29 & 30 & 31 & VSR[XT]
\end{array}
\]

\[
\begin{align*}
XT & \leftarrow TX \ || \ T \\
XA & \leftarrow AX \ || \ A \\
XB & \leftarrow BX \ || \ B \\
ex_flag & \leftarrow 0b0 \\
all_false & \leftarrow 0b1 \\
all_true & \leftarrow 0b1
\end{align*}
\]

do \ i=0 \ to \ 127 \ by \ 64 \\
reset_xflags() \\
src1 & \leftarrow \text{VSR>[XA]}{[i:i+63]} \\
src2 & \leftarrow \text{VSR>[XB]}{[i:i+63]} \\
\text{vxsnan_flag} & \leftarrow \text{IsSNaN}(src1) \ | \ \text{IsSNaN}(src2)
\]

\[
\begin{align*}
\text{if( CompareEQDP(src1,src2) ) then} \\
\text{result}{[i:i+63]} & \leftarrow 0xFFFF_FFFF_FFFF_FFFF \\
all_false & \leftarrow 0b0 \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{else do} \\
\text{result}{[i:i+63]} & \leftarrow 0x0000_0000_0000_0000 \\
all_true & \leftarrow 0b0 \\
\text{end} \\
\text{if(\ vxsnan_flag) then SetFX(VXSNAN) } \\
ex_flag & \leftarrow ex_flag \ | \ \text{ex_flag & vxsnan_flag}
\end{align*}
\]

\[
\text{if( ex_flag = 0 ) then VSR[XT] } \leftarrow \text{result}
\]

\[
\text{if(Rc=1) then do} \\
\text{if( !vex_flag ) then} \\
\text{CR[6]} & \leftarrow \text{all_true} \ || \ 0b0 \ || \ \text{all_false} \ || \ 0b0 \\
\text{else} \\
\text{CR[6]} & \leftarrow 0bUUUU \\
\text{end}
\]

\text{Let XT be the value TX concatenated with T.} \\
\text{Let XA be the value AX concatenated with A.} \\
\text{Let XB be the value BX concatenated with B.}

\begin{itemize}
\item For each vector element i from 0 to 1, do the following.
\item Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
\item Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
\item src1 is compared to src2.
\item The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.
\item A NaN input causes the comparison to return false for that element.
\end{itemize}

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.
\begin{itemize}
\item Bit 0 of CR[6] is set to indicate all vector elements compared true.
\item Bit 1 of CR[6] is set to 0.
\item Bit 2 of CR[6] is set to indicate all vector elements compared false.
\item Bit 3 of CR[6] is set to 0.
\end{itemize}

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

\textbf{Special Registers Altered}

\begin{itemize}
\item CR[6] \hspace{2cm} \text{.................................} \ (if Rc=1)
\item FX \hspace{2cm} VXSNAN
\end{itemize}

\textbf{VSR Data Layout for xvcmpeqdp[.]} \\
src1 = VSR[XA] \\
\begin{array}{cccc}
\text{DP} & \text{DP} \\
0 & 64 & 127
\end{array}
\]

src2 = VSR[XB] \\
\begin{array}{cccc}
\text{DP} & \text{DP} \\
0 & 64 & 127
\end{array}
\]

tgt = VSR[XT] \\
\begin{array}{cccc}
\text{MD} & \text{MD} \\
0 & 64 & 127
\end{array}
\]
**VSX Vector Compare Equal To Single-Precision [ & Record ] XX3-form**

\[
xvcmpeqsp\ XT,XA,XB \quad (Rc=0) \\
xvcmpeqsp.\ XT,XA,XB \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th>X</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>67</th>
<th>68</th>
<th>69</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←</td>
<td>TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>←</td>
<td>AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all_false</td>
<td>←</td>
<td>0b1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all_true</td>
<td>←</td>
<td>0b1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do i=0 to 127 by 32} \\
\text{reset_xflags()}
\]

\[
\text{src1} \leftarrow \text{VSR[XA]}_{i:i+31} \\
\text{src2} \leftarrow \text{VSR[XB]}_{i:i+31} \\
\text{vxsnan_flag} \leftarrow \text{IsSNaN(src1)} \lor \text{IsSNaN(src2)}
\]

\[
\text{if( CompareEQSP(src1,src2) ) then} \\
\quad \text{result}_{i:i+31} \leftarrow 0xFFFF_FFFF \\
\quad \text{all_false} \leftarrow 0b0 \\
\text{end} \\
\text{else do} \\
\quad \text{result}_{i:i+31} \leftarrow 0x0000_0000 \\
\quad \text{all_true} \leftarrow 0b0 \\
\text{end}
\]

\[
\text{if(vxsnan_flag) then SetFX(VXSNAN)} \\
\quad \text{ex_flag} \leftarrow \text{ex_flag} \lor (\text{VE} \lor \text{vxsnan_flag})
\]

\[
\text{if( ex_flag = 0 ) then VSR[XT] \leftarrow \text{result}}
\]

**Let XT be the value TX concatenated with T.**
**Let XA be the value AX concatenated with A.**
**Let XB be the value BX concatenated with B.**

For each vector element i from 0 to 3, do the following.

**Let src1 be the single-precision floating-point operand in word element i of VSR[XA].**

**Let src2 be the single-precision floating-point operand in word element i of VSR[XB].**

**src1 is compared to src2.**

The contents of word element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

**Special Registers Altered**

\[
\text{CR}[6] \\
\text{FX VXSNAN}
\]

**VSR Data Layout for xvcmpeqsp[.]**

\[
\text{src1} = \text{VSR[XA]} \\
\text{src2} = \text{VSR[XB]} \\
\text{tgt} = \text{VSR[XT]} \\
\]

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>MW</td>
<td>MW</td>
<td>MW</td>
<td>MW</td>
<td>MW</td>
</tr>
</tbody>
</table>
VSX Vector Compare Greater Than or Equal To Double-Precision [ & Record ] XX3-form

xvcmpgedp XT,XA,XB (Rc=0)
xvcmpgedp. XT,XA,XB (Rc=1)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>115</td>
</tr>
</tbody>
</table>

XT ← TX || T
XA ← AX || A
XB ← BX || B
ex_flag ← 0b0
all_false ← 0b1
all_true ← 0b1

do i=0 to 127 by 64
reset_xflags();
src1 ← VSR[XA][i:i+63]
src2 ← VSR[XB][i:i+63]
if( IsSNaN(src1) | IsSNaN(src2) ) then do
vxsnan_flag ← 0b1
if(VE=0) then vxvc_flag ← 0b1
end
else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)
if( CompareGEDP(src1,src2) ) then
result[i:i+63] ← 0xFFFF_FFFF_FFFF_FFFF
all_false ← 0b0
end
else do
result[i:i+63] ← 0x0000_0000_0000_0000
all_true ← 0b0
end
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxvc_flag) then SetFX(VXVC)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxvc_flag)
end
if( ex_flag = 0 ) then VSR[XT] ← result
if(Rc=1) then do
if( !vex_flag ) then
CR[6] ← all_true || 0b0 || all_false || 0b0
else
CR[6] ← 0bUUUU
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.
VSX Vector Compare Greater Than or Equal To Single-Precision [ & record CR6 ]

XX3-form

xvcmpgesp XT,XA,XB (Rc=0)

xvcmpgesp XT,XA,XB (Rc=1)

| 60 | 6 | T | A | B | Rc | 83 | X8B
|----|---|---|---|---|----|----|-----
|    |   |   |   |   |    | 63 | 63 |

XT ← TX || T
XA ← AX || A
XB ← BX || B

ex_flag ← 0b0
all_false ← 0b1
all_true ← 0b1

doi=0 to 127 by 32
reset_xflags()

src1 ← VSR[XA]{i:i+31}
src2 ← VSR[XB]{i:i+31}

if( IsSNaN(src1) | IsSNaN(src2) ) then do
vxsnan_flag ← 0b1
if(VE=0) then vxvc_flag ← 0b1
end
else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

if( CompareGESP(src1,src2) ) then
result{i:i+31} ← 0xFFFF_FFFF
all_false ← 0b0
end
else do
result{i:i+31} ← 0x0000_0000
all_true ← 0b0
end
if( !vex_flag ) then
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
if( !vex_flag ) then
CR[6] ← all_true || 0b0 || all_false || 0b0
else
CR[6] ← 0bUUUU
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than or equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.
– Bit 0 of CR[6] is set to indicate all vector elements compared true.
– Bit 1 of CR[6] is set to 0.
– Bit 2 of CR[6] is set to indicate all vector elements compared false.
– Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

CR[6] . . . . . . . . . . . . . . . . . . . . . . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgesp[.]

src1 = VSR[XA]
src2 = VSR[XB]
VSX Vector Compare Greater Than Double-Precision [ & record CR6 ] XX3-form

xvcmpgtdp XT,XA,XB (Rc=0)
xvcmpgtdp XT,XA,XB (Rc=1)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

XT ← TX || T
XA ← AX || A
XB ← BX || B
ex_flag ← 0b0
all_false ← 0b1
all_true ← 0b1

do i=0 to 127 by 64
reset_xflags()
srcl ← VSR[XA] {i:i+63}
srcl2 ← VSR[XB] {i:i+63}

if( IsSNAN(srcl) | IsSNAN(srcl2) ) then do
vxsnan_flag ← 0b1
if(VE=0) then vxvc_flag ← 0b1
end
else vxvc_flag ← IsQNaN(srcl) | IsQNaN(srcl2)

if( CompareGTDP(srcl,srcl2) ) then do
result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
all_false ← 0b0
end
else do
result{i:i+63} ← 0x0000_0000_0000_0000
all_true ← 0b0
end
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxvc_flag) then SetFX(VXVC)
ex_flag ← ex_flag || (VE & vxsnan_flag)
ex_flag ← ex_flag || (VE & vxvc_flag)
end
if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
if( !vex_flag ) then
CR[6] ← all_true || 0b0 || all_false || 0b0
else
CR[6] ← 0bUUUU
end

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let srcl be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.
VSX Vector Compare Greater Than Single-Precision [ & record CR6 ] XX3-form

\[
xvcmpgtsp \quad XT,XA,XB \quad (Rc=0) \\
xvcmpgtsp. \quad XT,XA,XB \quad (Rc=1)
\]

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>56</th>
<th>52</th>
<th>48</th>
<th>44</th>
<th>40</th>
<th>36</th>
<th>32</th>
<th>28</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>08</th>
<th>04</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>Rc</td>
<td>75</td>
<td>60</td>
<td>48</td>
<td>36</td>
<td>24</td>
<td>12</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
XT \leftarrow TX \| T \\
XA \leftarrow AX \| A \\
XB \leftarrow BX \| B \\
ex\_flag \leftarrow 0b0 \\
all\_false \leftarrow 0b1 \\
all\_true \leftarrow 0b1
\]

do i=0 to 127 by 32

reset_xflags()

src1 \leftarrow VSR[AX][i:i+31]

src2 \leftarrow VSR[BX][i:i+31]

if( IsSNaN(src1) | IsSNaN(src2) ) then do

vxsnan_flag \leftarrow 0b1

if(VE=0) then vxvc_flag \leftarrow 0b1

end

else vxvc_flag \leftarrow IsQNaN(src1) | IsQNaN(src2)

if( CompareGTSP(src1,src2) ) then do

result[i:i+31] \leftarrow 0xFFFF_FFFF

all_false \leftarrow 0b0

end

else do

result[i:i+31] \leftarrow 0x0000_0000

all_true \leftarrow 0b0

end

if(vxsnan_flag) then SetFX(VXSNAN)

if(vxvc_flag) then SetFX(VXVC)

ex_flag \leftarrow ex_flag | (VE & vxsnan_flag)

ex_flag \leftarrow ex_flag | (VE & vxvc_flag)

end

if( ex_flag = 0 ) then VSR[XT] \leftarrow result

if(Rc=1) then do

if( !vex_flag ) then

CR[6] \leftarrow all_true || 0b0 || all_false || 0b0

else

CR[6] \leftarrow 0bUUUU

end

Let XT be the value TX concatenated with T.

Let XA be the value AX concatenated with A.

Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

**Special Registers Altered**

CR[6] .............................. (if Rc=1)

FX VXSNAN VXVC

---

\[
\text{VSR Data Layout for xvcmpgtsp[.]} \\
\]

src1 = VSR[XA]

\[
\begin{array}{c|c|c|c|c}
\hline
& SP & SP & SP & SP \\
\hline
0 & 32 & 64 & 96 & 127 \\
\hline
\end{array}
\]

src2 = VSR[XB]

\[
\begin{array}{c|c|c|c|c}
\hline
& SP & SP & SP & SP \\
\hline
0 & 32 & 64 & 96 & 127 \\
\hline
\end{array}
\]

tgt = VSR[XT]

\[
\begin{array}{c|c|c|c|c}
\hline
& MW & MW & MW & MW \\
\hline
0 & 32 & 64 & 96 & 127 \\
\hline
\end{array}
\]
VSX Vector Copy Sign Double-Precision
XX3-form

**xvcpsgndp XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>240</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
&\text{XT} \leftarrow \text{TX} \ || \ T \\
&\text{XA} \leftarrow \text{AX} \ || \ A \\
&\text{XB} \leftarrow \text{BX} \ || \ B \\
&\text{do \ i=0 to 127 by 64} \\
&\hspace{1em} \text{VSR}[\text{XT}]_{i:i+63} \leftarrow \text{VSR}[\text{XA}]_{i} || \text{VSR}[\text{XB}]_{i+1:i+63} \\
&\text{end}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
The contents of bit 0 of doubleword element i of VSR[XA] are concatenated with the contents of bits 1:63 of doubleword element i of VSR[XB] and placed into doubleword element i of VSR[XT].

**Special Registers Altered**
None

**Extended Mnemonic Equivalent To**
```
xvmovdp XT,XB
xvcpsgndp XT,XB,XB
```

**Table 82: VSR Data Layout for xvcpsgndp**

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>src2 = VSR[XB]</th>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td>DP</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>127</td>
</tr>
</tbody>
</table>

VSX Vector Copy Sign Single-Precision
XX3-form

**xvcpsgnsp XT,XA,XB**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>208</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
&\text{XT} \leftarrow \text{TX} \ || \ T \\
&\text{XA} \leftarrow \text{AX} \ || \ A \\
&\text{XB} \leftarrow \text{BX} \ || \ B \\
&\text{do \ i=0 to 127 by 32} \\
&\hspace{1em} \text{VSR}[\text{XT}]_{i:i+31} \leftarrow \text{VSR}[\text{XA}]_{i} || \text{VSR}[\text{XB}]_{i+1:i+31} \\
&\text{end}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
The contents of bit 0 of word element i of VSR[XA] are concatenated with the contents of bits 1:31 of word element i of VSR[XB] and placed into word element i of VSR[XT].

**Special Registers Altered**
None

**Extended Mnemonic Equivalent To**
```
xvmovsp XT,XB
xvcpsgnsp XT,XB,XB
```

**Table 83: VSR Data Layout for xvcpsgnsp**

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>src2 = VSR[XB]</th>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>
VSX Vector round Double-Precision to single-precision and Convert to Single-Precision format XX2-form

```
xvcvdpsp   XT, XB
   60   T  ///  B   393  393
   0   6 11 16   31  32  63
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN

VSR Data Layout for xvcvdpsp

```
src = VSR[XB]
tgt = VSR[XT]
```

```
VSX Vector truncate Double-Precision to Integer and Convert to Signed Integer Doubleword format with Saturate XX2-form

\[
\text{\texttt{xvcvdpsxds \ XT, XB}}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>53</th>
<th>52</th>
<th>51</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT ← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB ← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag ← 000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{do } i &= 0 \text{ to } 127 \text{ by } 64 \\
&\text{reset_xflags();} \\
&\text{rnd}(i:63) ← \text{RoundToDPIntegerTrunc(VSR[XB][i:i+63])} \\
&\text{result}(i:i+63) ← \text{ConvertDPtoSD(rnd)} \\
&\text{if(vxsnan_flag) then SetFX(VXSNAN)} \\
&\text{if(vxcvi_flag) then SetFX(VXCVI)} \\
&\text{if(xx_flag) then SetFX(XX)} \\
&\text{ex_flag ← ex_flag | (VE & vxsnan_flag)} \\
&\text{ex_flag ← ex_flag | (VE & vxcvi_flag)} \\
&\text{ex_flag ← ex_flag | (XE & xx_flag)}
\end{align*}
\]

\[
\text{end}
\]

\[
\text{if( ex_flag = 0 ) then VSR[XT] ← result}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let \( src \) be the double-precision floating-point operand in doubleword element i of VSR[XB].

If \( src \) is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If \( src \) is an SNaN, VXSNAN is also set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{63}-1, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

The result is placed into doubleword element i of VSR[XT].

See Table 84.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

| FX | XX | VXSNAN | VXCVI |

VSR Data Layout for xvcvdpsxds

<table>
<thead>
<tr>
<th>( \text{src} = \text{VSR[XB]} )</th>
<th>( \text{tgt} = \text{VSR[XT]} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
<tr>
<td>SD</td>
<td>SD</td>
</tr>
</tbody>
</table>

Programming Note

\textbf{xvcvdpsxds} rounds using Round towards Zero rounding mode. For other rounding modes, software must use a \textit{Round to Double-Precision Integer} instruction that corresponds to the desired rounding mode, including \textbf{xvrdpic} which uses the rounding mode specified by the RN field in the FPSCR.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 497
VSX Vector truncate Double-Precision to Integer and Convert to Signed Integer Word format with Saturate XX2-form

xvcvdpsxws XT,XB

<table>
<thead>
<tr>
<th>xvcvdpsxws</th>
<th>XT</th>
<th>XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>216</td>
<td></td>
</tr>
</tbody>
</table>

- Let XT be the value TX concatenated with T.
- Let XB be the value BX concatenated with B.

For each vector element i from 0 to 127 by 64, do the following.

Let \( src \) be the double-precision floating-point operand in doubleword element i of VSR[XB].

If \( src \) is a NaN, the result is the value 0x8000_0000 and VXCVI is set to 1. If \( src \) is an SNaN, VXSNAN is also set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \( 2^{31} - 1 \), the result is 0x7FFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than \( -2^{31} \), the result is 0x8000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

The result is placed into bits 0:31 of doubleword element i of VSR[XT].

The contents of bits 32:63 of doubleword element 1 of VSR[XT] are undefined.

See Table 85.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

<table>
<thead>
<tr>
<th>FX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXCVI</th>
</tr>
</thead>
</table>

VSR Data Layout for xvcvdpsxws

<table>
<thead>
<tr>
<th>xvcvdpsxws</th>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SW</td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<table>
<thead>
<tr>
<th>tgt</th>
<th>undefined</th>
<th>undefined</th>
</tr>
</thead>
</table>

Programming Note

xvcvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Double-Precision Integer instruction that corresponds to the desired rounding mode, including xvrdpic which uses the rounding mode specified by the RN field in the FPSCR.
## Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>0</th>
<th>-</th>
<th>T(Nmin), fx(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0 yes</td>
<td>T(Nmin), fx(XX)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 yes</td>
<td>fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>-</td>
<td>-</td>
<td>no T(Nmin)</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>- no</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src)))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 yes</td>
<td>T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), fx(XX)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 yes</td>
<td>fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>-</td>
<td>-</td>
<td>no T(Nmax)</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>0 yes</td>
<td>T(Nmax), fx(XX)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 yes</td>
<td>T(Nmax), fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src ≥ Nmax+1</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmax), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
</tr>
<tr>
<td>sc is a QNaN</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmin), fx(VXCVI)</td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>fx(VXCVI), error()</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0 -</td>
<td>-</td>
<td>T(Nmin), fx(VXCVI), fx(VXSNAN)</td>
</tr>
<tr>
<td></td>
<td>1 -</td>
<td>-</td>
<td>fx(VXCVI), fx(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:
- ConvertDPtoSW(x) The double-precision floating-point integer value x converted to signed integer word format.
- fx(x) FX is set to 1 if x=0. x is set to 1.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- Nmin The smallest signed integer word value, \(-2^{31}(0x8000_0000)\).
- Nmax The largest signed integer word value, \(2^{31}-1 (0x7FFFF_FFFF)\).
- RoundToDPintegerTrunc(x) The double-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- src The double-precision floating-point value in doubleword element i of VSR[XB] (where i \(\in (0,1)\)).
- T(x) The signed integer word value x is placed in word element i of VSR[XT] (where i \(\in (0,2)\)).

### Table 85. Actions for xcvdpsxws
VSX Vector truncate Double-Precision to Integer and Convert to Unsigned Integer Doubleword format with Saturate \( XX2 \)-form

\[
xvcvdpuxds \quad XT, XB
\]

<table>
<thead>
<tr>
<th>( i )</th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>456</th>
<th>B</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>/ / /</td>
<td>B</td>
<td>456</td>
<td>B</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

\[
XT \leftarrow TX || T
XB \leftarrow BX || B
ex\_flag \leftarrow 0b0
\]

\[
do\ i=0\ to\ 127\ by\ 64
\]
\[
\begin{align*}
\text{reset\_xflags();} \\
\text{rnd}(0:63) \leftarrow \text{RoundToDIPIntegerTrunc}(\text{VSR}[XB](i:i+63)) \\
\text{result}(i:i+63) \leftarrow \text{ConvertDpToUD}(\text{rnd}) \\
\text{if(vxsnan\_flag) then SetFX(VXSNAN)} \\
\text{if(vxcvi\_flag) then SetFX(VXCVI)} \\
\text{if(xx\_flag) then SetFX(XX)} \\
\text{ex\_flag} \leftarrow \text{ex\_flag} \land (\text{VE} \land \text{vxsnan\_flag}) \\
\text{ex\_flag} \leftarrow \text{ex\_flag} \land (\text{VE} \land \text{vxcvi\_flag}) \\
\text{ex\_flag} \leftarrow \text{ex\_flag} \land (\text{XE} \land \text{xx\_flag})
\end{align*}
\]

\[
\text{end}
\]

\[
\text{if( ex\_flag = 0 ) then VSR}[XT] \leftarrow \text{result}
\]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

For each vector element \( i \) from 0 to 1, do the following.

Let \( src \) be the double-precision floating-point operand in doubleword element \( i \) of \( \text{VSR}[XB] \).

If \( src \) is a NaN, the result is the value \( 0x0000_0000_0000_0000 \) and VXCVI is set to 1. If \( src \) is an SNaN, VXSNAN is also set to 1.

Otherwise, \( src \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \( 2^{64} - 1 \), the result is \( 0xFFFF_FFFF_FFFF_FFFF \) and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is \( 0x0000_0000_0000_0000 \) and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

The result is placed into doubleword element \( i \) of \( \text{VSR}[XT] \).

See Table 86.

If a trap-enabled exception occurs in any element of the vector, no results are written to \( \text{VSR}[XT] \).
### Returned Results and Status Setting

| src ≤ Nmin-1 | 0  | – | T(Nmin), fx(VXCVI) |  
| src ≤ Nmin-1 | 1  | – | fx(VXCVI), error() |
| Nmin-1 < src < Nmin | 0  | yes | T(Nmin), fx(XX) | 
| Nmin-1 < src < Nmin | 1  | yes | fx(XX), error() |
| src = Nmin | –  | no | T(Nmin) |
| Nmin < src < Nmax | –  | no | T(ConvertDPtoUD(RoundToDPintegerTrunc(src))) |
| Nmin < src < Nmax | 0  | yes | T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), fx(XX) |
| Nmin < src < Nmax | 1  | yes | fx(XX), error() |
| src = Nmax | –  | no | T(Nmax) |
| Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness. |
| Nmin < src < Nmax+1 | –  | no | T(Nmax) |
| Nmin < src < Nmax+1 | 0  | yes | T(Nmax), fx(XX) |
| Nmin < src < Nmax+1 | 1  | yes | T(Nmax), fx(XX), error() |
| src ≤ Nmax+1 | 0  | – | T(Nmax), fx(VXCVI) |
| src ≤ Nmax+1 | 1  | – | fx(VXCVI), error() |
| src is a QNaN | 0  | – | T(Nmax), fx(VXCVI), fx(VXSNAN) |
| src is a QNaN | 1  | – | fx(VXCVI), fx(VXSNAN), error() |

**Explanation:**
- ConvertDPtoUD(x): The double-precision floating-point integer value x converted to unsigned integer doubleword format.
- fx(x): FX is set to 1 if x=0. x is set to 1.
- error(): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- Nmin: The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
- Nmax: The largest unsigned integer doubleword value, 2^{64} - 1 (0xFF_FFFF_FFFF_FFFF).
- RoundToDPintegerTrunc(x): The double-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- src: The double-precision floating-point value in doubleword element i VSR[XB] (where i ∈ (0,1)).
- T(x): The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ (0,1)).

Table 86: Actions for xvcvdpxuds
VSX Vector truncate Double-Precision to Integer and Convert to Unsigned Integer Word format with Saturate XX2-form

\[
\begin{align*}
\text{xtcvcdpuxws} & \quad \text{XT, XB} \\
\hline
& 60 & 6 & 11 & 16 & 21 & 200 & 0 \times 8 \\
\text{XT} & \leftarrow & \text{TX} && & & & \\
\text{XB} & \leftarrow & \text{BX} && & & & \\
\text{ex\_flag} & \leftarrow & 0b0 & & & & \\
\end{align*}
\]

\[
\begin{align*}
d & \text{do i=0 to 127 by 64} \\
& \text{reset\_xflags()} \\
& \text{rnd}(0:63) \leftarrow \text{RoundToIntegerTrunc(VSR[XB](i:i+63))} \\
& \text{result}(i:i+31) \leftarrow \text{ConvertDPtoUW(rnd)} \\
& \text{if(vxsnan\_flag) then SetFX(VXSNAN)} \\
& \text{if(vxcvi\_flag) then SetFX(VXCVI)} \\
& \text{if(xx\_flag) then SetFX(XX)} \\
& \text{ex\_flag} \leftarrow \text{ex\_flag | (VE & vxsnan\_flag)} \\
& \text{ex\_flag} \leftarrow \text{ex\_flag | (VE & vxcvi\_flag)} \\
& \text{ex\_flag} \leftarrow \text{ex\_flag | (XE & xx\_flag)} \\
& \text{end} \\
& \text{if( ex\_flag = 0 ) then VSR[XT] \leftarrow result}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \(2^{32} - 1\), the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

The result is placed into bits 0:31 of doubleword element i of VSR[XT].

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

See Table 87.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>VE</th>
<th>XE</th>
<th>Inexact (RoundToDPIntegerTrunc(src)) = src</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ Nmin-1</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T(Nmin), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>–</td>
<td>no</td>
<td>T(ConvertDPtoUW(RoundToDPIntegerTrunc(src)))</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(ConvertDPtoUW(RoundToDPIntegerTrunc(src))), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>–</td>
<td>0 yes</td>
<td>T(Nmax), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1 yes</td>
<td>fx(XX), error()</td>
<td></td>
</tr>
<tr>
<td>src ≥ Nmax+1</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

**Explanation:**

- ConvertDPtoUW(x) The double-precision floating-point integer value x converted to unsigned integer word format.
- fx(x) FX is set to 1 if x=0. x is set to 1.
- error() The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- Nmin The smallest unsigned integer word value, 0 (0x0000_0000).
- Nmax The largest unsigned integer word value, $2^{32} - 1$ (0xFFFF_FFFF).
- RoundToDPIntegerTrunc(x) The double-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- src The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0, 1}).
- T(x) The unsigned integer word value x is placed in word element i of VSR[XT] (where i ∈ {0, 2}).

Table 87. Actions for xvcvdpuxws


VSX Vector Convert Single-Precision to Double-Precision format *XX2-form*

\[
xvcvspdp \quad XT,XB
\]

\[
\begin{array}{cccccc}
0 & T & \text{//} & B & 457 & \text{BITX}\\
6 & 11 & 16 & 21 & 30 & 31
\end{array}
\]

\[
\begin{align*}
XT & \leftarrow TX \ || \ T \\
XB & \leftarrow BX \ || \ B \\
ex_{\text{flag}} & \leftarrow 0b0 \\
do i=0 \text{ to } 127 \text{ by } 64 \\
& \quad \text{reset_xflags()}
\quad \text{result}_{i:i+63} \leftarrow \text{ConvertSPtoDP}(VSR[XB][i:i+31]) \\
& \quad \text{if}(\text{vxsnan_flag}) \text{ then SetFX(VXSNAN)} \\
& \quad \text{ex}_{\text{flag}} \leftarrow \text{ex}_{\text{flag}} \ | \ (\text{VE} \ &: \ \text{vxsnan_flag}) \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{if}( \ ex_{\text{flag}} = 0 ) & \text{ then VSR[XT]} \leftarrow \text{result}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let \( src \) be the single-precision floating-point operand in bits 0:31 of doubleword element i of VSR[XB].

\( src \) is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

\( \text{FX} \quad \text{VXSNAN} \)

**VSR Data Layout for xvcvspdp**

\( \text{src} = \text{VSR[XB]} \)

\[
\begin{array}{cccccc}
\text{SP} & \text{unused} & \text{SP} & \text{unused}
\end{array}
\]

\( \text{tgt} = \text{VSR[XT]} \)

\[
\begin{array}{cccccc}
\text{DP} & \text{DP}
\end{array}
\]

0 32 64 96 127
VSX Vector truncate Single-Precision to integer and Convert to Signed Integer
Doubleword format with Saturate  XX-form

```
xt|xb|ex_flag
-----------------
0|6|11|16|21|408|60
```

```
XT ← TX || T
XB ← BX || B
ex_flag ← 0b0
```

do i=0 to 127 by 64
  reset_xflags()
  rnd{i:i+31} ← RoundToSPIntegerTrunc(VSR[XB]{i:i+31})
  result{i:i+63} ← ConvertSPtoSD(rnd)
  if(vxsnan_flag) then SetFX(VXSNAN)
  if(vxcvi_flag)  then SetFX(VXCVI)
  if(xx_flag)     then SetFX(XX)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
  ex_flag ← ex_flag | (VE & vxcvi_flag)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
  Let src be the single-precision floating-point operand in word element i×2 of VSR[XB].
  If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.
  Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.
  If the rounded value is greater than 2^{63}-1, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.
  Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.
  Otherwise, the result is the rounded value converted to 64-bit signed-integer format.
  The result is placed into doubleword element i of VSR[XT].

See Table 87.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src</th>
<th>NE</th>
<th>XE</th>
<th>Impact? {RoundToSPIntegerTrunc(src) == src}</th>
</tr>
</thead>
<tbody>
<tr>
<td>src ≤ Nmin-1</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>–</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmin</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>yes</td>
<td>T(ConvertSPtoSD(RoundToSPIntegerTrunc(src))), fx(XX)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src = Nmax</td>
<td>–</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>–</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>fx(XX), error()</td>
</tr>
<tr>
<td>src ≥ Nmax+1</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

### Explanation:

- **ConvertSPtoSD(x)**: The single-precision floating-point integer value x converted to signed integer doubleword format.
- **fx(x)**: FX is set to 1 if x=0. x is set to 1.
- **error()**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- **Update of VSR[XT] is suppressed.**
- **Nmin**: The smallest signed integer doubleword value, $-2^{63}$ (0x8000_0000_0000_0000).
- **Nmax**: The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
- **RoundToSPIntegerTrunc(x)**: The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- **src**: The single-precision floating-point value in word element i of VSR[XB] (where i ∈ (0,2)).
- **T(x)**: The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ (0,1)).

### Table 88. Actions for xcvspxsdxds
VSX Vector truncate Single-Precision to integer and Convert to Signed Integer Word format with Saturate X2-form

- **xvcvspsxws**
- **XT, XB**

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>152</th>
<th>B[1:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Let XT be the value TX concatenated with T.
- Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

#### Programming Note

**xvcvspsxws** rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including **xvrspic** which uses the rounding mode specified by the RN field in the FPSCR.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

--

Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 507
Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>NE</th>
<th>NE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>T(Nmin), lx(VXCVI)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>lx(VXCVI), error()</td>
</tr>
</tbody>
</table>

Nmin-1 < src < Nmin

| -  | 0 | yes | T(Nmin), lx(XX) |
| 1 | yes | lx(XX), error() |

src = Nmin

| -  | no | T(Nmin) |

Nmin < src < Nmax

| -  | no | T(ConvertSPtoSW(RoundToSPintegerTrunc(src))) |
| 0 | yes | T(ConvertSPtoSW(RoundToSPintegerTrunc(src))), lx(XX) |
| 1 | yes | lx(XX), error() |

src = Nmax

| -  | no | T(Nmax) |

Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.

Nmax < src < Nmax+1

| -  | 0 | yes | T(Nmax), lx(XX) |
| 1 | yes | lx(XX), error() |

src ≥ Nmax+1

| -  | -  | T(Nmax), lx(VXCVI) |
| 1 | -  | lx(VXCVI), error() |

src is a QNaN

| -  | -  | T(Nmin), lx(VXCVI) |
| 1 | -  | lx(VXCVI), error() |

src is a SNaN

| -  | -  | T(Nmin), lx(VXCVI), lx(VXSNAN) |
| 1 | -  | lx(VXCVI), lx(VXSNAN), error() |

Explanation:

- **ConvertSPtoSW(x)**: The single-precision floating-point integer value x converted to signed integer word format.
- **Tx(x)**: The signed integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
- **Tx(x)**: The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
- **Nmin**: The smallest signed integer word value, $-2^{31} (0x8000_0000)$. 
- **Nmax**: The largest signed integer word value, $2^{31} - 1 (0xFFFF_FFFF)$. 
- **RoundToSPintegerTrunc(x)**: The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- **error()**: The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).

Table 89. Actions for xcvpspxws
VSX Vector truncate Single-Precision to integer and Convert to Unsigned Integer Doubleword format with Saturate XX2-form

**xvcvspuxds**

<table>
<thead>
<tr>
<th>XT, XB</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>392</td>
</tr>
</tbody>
</table>

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 127, do the following.

Let src be the single-precision floating-point operand in word element i×2 of VSR[ XB].

If src is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

The result is placed into doubleword element i of VSR[ XT].

See Table 87.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[ XT].

**Programming Note**

**xvcvspuxds** rounds using Round towards Zero rounding mode. For other rounding modes, software must use a Round to Single-Precision Integer instruction that corresponds to the desired rounding mode, including **xvrsfic** which uses the rounding mode specified by the RN field in the FPSCR.

**VSR Data Layout for xvcvspuxds**

| src = VSR[ XB] |
|---|---|---|---|---|
| SP | unused | SP | unused |

tgt = VSR[ XT]

<table>
<thead>
<tr>
<th>UD</th>
<th>UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>127</td>
<td>127</td>
</tr>
</tbody>
</table>

**Special Registers Altered**

FX  XX  VXSNAN  VXCVI
Table 90. Actions for xvcvspuxds

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>NE</th>
<th>KE</th>
<th>Returned Results and Status Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – –</td>
<td>T(min), lx(VXCVI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 – –</td>
<td>lx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0 yes</td>
<td>T(min), lx(XX)</td>
<td></td>
</tr>
<tr>
<td>1 yes</td>
<td>lx(XX), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmin</td>
<td>– – no</td>
<td>T(min)</td>
<td></td>
</tr>
<tr>
<td>Nmin &lt; src &lt; Nmax</td>
<td>– – no</td>
<td>T(ConvertSPtoUD(RoundToSPIntegerTrunc(src)))</td>
<td></td>
</tr>
<tr>
<td>0 yes</td>
<td>T(ConvertSPtoUD(RoundToSPIntegerTrunc(src))), lx(XX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 yes</td>
<td>lx(XX), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src = Nmax</td>
<td>– – no</td>
<td>T(max)</td>
<td></td>
</tr>
<tr>
<td>Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nmax &lt; src &lt; Nmax+1</td>
<td>– 0 yes</td>
<td>T(max), lx(XX)</td>
<td></td>
</tr>
<tr>
<td>1 yes</td>
<td>lx(XX), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src ≥ Nmax+1</td>
<td>0 – –</td>
<td>T(max), lx(VXCVI)</td>
<td></td>
</tr>
<tr>
<td>1 – –</td>
<td>lx(VXCVI), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a QNaN</td>
<td>0 – –</td>
<td>T(min), lx(VXCVI), lx(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td>1 – –</td>
<td>lx(VXCVI), lx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src is a SNaN</td>
<td>0 – –</td>
<td>T(min), lx(VXCVI), lx(VXSNAN)</td>
<td></td>
</tr>
<tr>
<td>1 – –</td>
<td>lx(VXCVI), lx(VXSNAN), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

- ConvertSPtoUD(x): The single-precision floating-point integral value x converted to unsigned integer doubleword format.
- fx(x): FX is set to 1 if x=0, x is set to 1.
- error(): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
- Update of VSR[XT] is suppressed.
- Nmin: The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
- Nmax: The largest unsigned integer doubleword value, 2 64-1 (0xFFFF_FFFF_FFFF_FFFF).
- RoundToSPIntegerTrunc(x): The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- T(x): The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ (0,2)).
- lx(x): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.

510 Power ISA™ - Book I
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 511

VSX Vector truncate Single-Precision to integer and Convert to Unsigned Integer Word format with Saturate XX2-form

\[
\text{xtcvspxws} \quad \text{XT, XB}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>136</th>
<th>Bn</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XT</strong></td>
<td>←</td>
<td>TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XB</strong></td>
<td>←</td>
<td>BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ex_flag</strong></td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by 32} \\
\text{reset_xflags()} \\
\text{rnd}(i:i+31) \leftarrow \text{RoundToSPIntegerTrunc}(src) \\
\text{result}(i:i+31) \leftarrow \text{ConvertSPtoUW}(\text{rnd}) \\
\text{if(vxsnan_flag) then SetFX(VXSNAN) } \\
\text{if(vxcvi_flag) then SetFX(VXCVI) } \\
\text{if(xx_flag) then SetFX(XX) } \\
\text{ex_flag} \leftarrow \text{ex_flag} \lor (\text{VE} \lor \text{vxsnan_flag}) \\
\text{ex_flag} \leftarrow \text{ex_flag} \lor (\text{VE} \lor \text{vxcvi_flag}) \\
\text{ex_flag} \leftarrow \text{ex_flag} \lor (\text{XE} \lor \text{xx_flag}) \\
\text{end}
\]

\[
\text{if( ex_flag = 0 ) then VSR[XT] } \leftarrow \text{result}
\]

Let **XT** be the value TX concatenated with T.
Let **XB** be the value BX concatenated with B.

For each vector element \( i \) from 0 to 3, do the following.

Let \( \text{src} \) be the single-precision floating-point operand in word element \( i \) of VSR[**XB**].

If \( \text{src} \) is a NaN, the result is the value 0x0000_0000 and VXCVI is set to 1. If \( \text{src} \) is an SNaN, VXSNAN is also set to 1.

Otherwise, \( \text{src} \) is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than \( 2^{32}-1 \), the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

The result is placed into word element \( i \) of VSR[**XT**].

See Table 87.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[**XT**].
### Returned Results and Status Setting

<table>
<thead>
<tr>
<th>src ≤ Nmin-1</th>
<th>0</th>
<th>-</th>
<th>-</th>
<th>T[min], h(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>h(VXCVI), error()</td>
</tr>
<tr>
<td>Nmin-1 &lt; src &lt; Nmin</td>
<td>0</td>
<td>yes</td>
<td>T[min], h(XX)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>h(XX), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| src = Nmin | - | - | no | T[min] |

| Nmin < src < Nmax | - | no | T(ConvertSPtoUW(RoundToSPintegerTrunc(src))) |
|-------------------|----|----|-----|------------------|
| 0                 | yes | T(ConvertSPtoUW(RoundToSPintegerTrunc(src))), h(XX) |
| 1                 | yes | h(XX), error() |

| src = Nmax | - | - | no | T[max] |

**Note:** This case cannot occur as Nmax is not representable in SP format but is included here for completeness.

<table>
<thead>
<tr>
<th>Nmax &lt; src &lt; Nmax+1</th>
<th>-</th>
<th>0</th>
<th>yes</th>
<th>T[max], h(XX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>h(XX), error()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src ≥ Nmax+1</th>
<th>0</th>
<th>-</th>
<th>-</th>
<th>T[max], h(VXCVI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>h(VXCVI), error()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src is a QNaN</th>
<th>0</th>
<th>-</th>
<th>-</th>
<th>T[min], h(VXCVI), h(VXSNAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>h(VXCVI), h(VXSNAN), error()</td>
</tr>
</tbody>
</table>

### Explanation:
- ConvertSPtoUW(x): The single-precision floating-point integer value x converted to unsigned integer word format.
- fx(x): FX is set to 1 if x=0. x is set to 1.
- error(): The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
- Nmin: The smallest unsigned integer word value, 0 (0x0000_0000).
- Nmax: The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
- RoundToSPintegerTrunc(x): The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
- src: The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
- T(x): The unsigned integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,1,2,3}).

### Table 91. Actions for xvcvspuxws
VSX Vector Convert and round Signed Integer Doubleword to Double-Precision format

**XX2-form**

**xvcvxsddp**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>SD</td>
</tr>
</tbody>
</table>

**xt,xb**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>504</th>
<th>BxT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>513</td>
</tr>
</tbody>
</table>

\[
\text{XT} \leftarrow TX \ || \ T \\
\text{XB} \leftarrow BX \ || \ B \\
\text{ex\_flag} \leftarrow 0b0
\]

do i=0 to 127 by 64

\[
\text{reset\_xflags()}
\]

\[
\text{v(0:inf)} \leftarrow \text{ConvertSDtoFP(VSR[XB](i;i+63))}
\]

\[
\text{result}[i:i+63] \leftarrow \text{RoundToDP(RN,v)}
\]

if(xx\_flag) then SetFX(XX)

\[
\text{ex\_flag} \leftarrow \text{ex\_flag} | (XE \ & \ xx\_flag)
\]

end

\[
\text{if( ex\_flag = 0 ) then VSR[XT] \leftarrow \text{result}}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX XX

**VSR Data Layout for xvcvxsddp**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>undefined</td>
<td>SP</td>
</tr>
</tbody>
</table>

VSX Vector Convert and round Signed Integer Doubleword to Single-Precision format

**XX2-form**

**xvcvxsdsp**

<table>
<thead>
<tr>
<th>src</th>
<th>tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>SD</td>
</tr>
</tbody>
</table>

**xt,xb**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>440</th>
<th>BxT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>440</td>
</tr>
</tbody>
</table>

\[
\text{XT} \leftarrow TX \ || \ T \\
\text{XB} \leftarrow BX \ || \ B \\
\text{ex\_flag} \leftarrow 0b0
\]

do i=0 to 127 by 64

\[
\text{reset\_xflags()}
\]

\[
\text{v(0:inf)} \leftarrow \text{ConvertSDtoFP(VSR[XB](i;i+63))}
\]

\[
\text{result}[i:i+31] \leftarrow \text{RoundToSP(RN,v)}
\]

\[
\text{result}[i+32:i+63] \leftarrow 0xUUUUUUUUU
\]

if(xx\_flag) then SetFX(XX)

\[
\text{ex\_flag} \leftarrow \text{ex\_flag} | (XE \ & \ xx\_flag)
\]

end

\[
\text{if( ex\_flag = 0 ) then VSR[XT] \leftarrow \text{result}}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX XX

**VSR Data Layout for xvcvxsdsp**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>undefined</td>
<td>SP</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>
VSX Vector Convert Signed Integer Word to Double-Precision format   XX2-form

xvcvsxwdp  XT,XB

| 60 | T | /// | B | 248 | BTX |

XT ← TX || T
XB ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
  reset_xflags()
  v[0:inf] ← ConvertSMtoFP(VSR[XB][i:i+31])
  result[i:i+63] ← RoundToDP(RN,v)
  if(xx_flag) then SetFX(XX)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
  Let src be the signed integer in bits 0:31 of doubleword element i of VSR[XB].
  src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

  The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX  XX

VSR Data Layout for xvcvsxwdp

src = VSR[XB]

SW   unused   SW   unused

tgt = VSR[XT]

DP   DP

0  64  96  127

VSX Vector Convert and round Signed Integer Word to Single-Precision format   XX2-form

xvcvsxwsp  XT,XB

| 60 | T | /// | B | 184 | BTP |

XT ← TX || T
XB ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
  reset_xflags()
  v[0:inf] ← ConvertSMtoFP(VSR[XB][i:i+31])
  result[i:i+31] ← RoundToSP(RN,v)
  if(xx_flag) then SetFX(XX)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
  Let src be the signed integer in word element i of VSR[XB].
  src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

  The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX  XX

VSR Data Layout for xvcvsxwsp

src = VSR[XB]

SW   SW   SW   SW

tgt = VSR[XT]

SP   SP   SP   SP

0  32  64  96  127
VSX Vector Convert and round Unsigned Integer Doubleword to Double-Precision format \(XX2\)-form

\[\text{xvcvuxddp} \quad \text{XT, XB}\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ / /</th>
<th>B</th>
<th>488</th>
<th>80x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>80</td>
</tr>
</tbody>
</table>

\[\begin{align*}
\text{XT} & \leftarrow \text{TX} \mathbin{\|} T \\
\text{XB} & \leftarrow \text{BX} \mathbin{\|} B \\
\text{ex\_flag} & \leftarrow 0b0
\end{align*}\]

do \text{i} = 0 to 127 by 64
reset_xflags()
\[\begin{align*}
\text{v}(\text{i}:\text{i}+63) & \leftarrow \text{ConvertUDtoFP(VSR[\text{XB}](\text{i}:\text{i}+63))} \\
\text{result}(\text{i}:\text{i}+63) & \leftarrow \text{RoundToDP(RN, v)} \\
\text{if(xx\_flag)} & \text{then SetFX(XX)} \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} \mathbin{\|} (\text{XE} \mathbin{\&} \text{xx\_flag})
\end{align*}\]
end

if(ex\_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element \(\text{i}\) from 0 to 1, do the following.

Let src be the unsigned integer in doubleword element \(\text{i}\) of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element \(\text{i}\) of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

\[\text{FX} \quad \text{XX}\]

VSR Data Layout for xvcvuxddp

\[\begin{array}{c|c}
\text{src} & \text{VSR[XB]} \\
\hline
\text{UD} & \text{UD} \\
\end{array}\]

\[\begin{array}{c|c}
\text{tgt} & \text{VSR[XT]} \\
\hline
\text{DP} & \text{DP} \\
\end{array}\]

VSX Vector Convert and round Unsigned Integer Doubleword to Single-Precision format \(XX2\)-form

\[\text{xvcvuxdsp} \quad \text{XT, XB}\]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ / /</th>
<th>B</th>
<th>424</th>
<th>80x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>80</td>
</tr>
</tbody>
</table>

\[\begin{align*}
\text{XT} & \leftarrow \text{TX} \mathbin{\|} T \\
\text{XB} & \leftarrow \text{BX} \mathbin{\|} B \\
\text{ex\_flag} & \leftarrow 0b0
\end{align*}\]

do \text{i} = 0 to 127 by 64
reset_xflags()
\[\begin{align*}
\text{v}(\text{i}:\text{i}+63) & \leftarrow \text{ConvertUDtoFP(VSR[\text{XB}](\text{i}:\text{i}+63))} \\
\text{result}(\text{i}:\text{i}+31) & \leftarrow \text{RoundToSP(RN, v)} \\
\text{result}(\text{i}+32:\text{i}+63) & \leftarrow 0xUUUU_UUUU \\
\text{if(xx\_flag)} & \text{then SetFX(XX)} \\
\text{ex\_flag} & \leftarrow \text{ex\_flag} \mathbin{\|} (\text{XE} \mathbin{\&} \text{xx\_flag})
\end{align*}\]
end

if(ex\_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element \(\text{i}\) from 0 to 1, do the following.

Let src be the unsigned integer in doubleword element \(\text{i}\) of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into bits 0:31 of doubleword element \(\text{i}\) of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element \(\text{i}\) of VSR[XT] are undefined.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

\[\text{FX} \quad \text{XX}\]

VSR Data Layout for xvcvuxdsp

\[\begin{array}{c|c}
\text{src} & \text{VSR[ XB]} \\
\hline
\text{UD} & \text{UD} \\
\end{array}\]

\[\begin{array}{c|c}
\text{tgt} & \text{VSR[ XT]} \\
\hline
\text{SP} & \text{undefined} \\
\end{array}\]

\[\begin{array}{c|c|c|c|c}
0 & 32 & 64 & 96 & 127 \\
\hline
\text{SP} & \text{undefined} & \text{SP} & \text{undefined} \\
\end{array}\]
VSX Vector Convert and round Unsigned Integer Word to Double-Precision format

**XX2-form**

**xvcvuxwdp**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>232</th>
<th>8X</th>
</tr>
</thead>
</table>

**XT, XB**

- **XT** ← **TX** || **T**
- **XB** ← **BX** || **B**
- **ex_flag** ← 0b0

- do i=0 to 127 by 64
  - reset_xflags(!)
  - **v[0:inf]** ← ConvertUWtoFP(VSR[XB][i:i+31])
  - **result[i:i+63]** ← RoundToDP(RN,v)
  - if(xx_flag) then SetFX(XX)
  - **ex_flag** ← **ex_flag** | (XE & xx_flag)

- if( ex_flag = 0 ) then VSR[XT] ← result

Let **XT** be the value **TX** concatenated with **T**.
Let **XB** be the value **BX** concatenated with **B**.

For each vector element **i** from 0 to 127, do the following.
Let **src** be the unsigned integer in bits 0:31 of doubleword element **i** of VSR[**XB**].

**src** is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field **RN** of the FPSCR.

The result is placed into doubleword element **i** of VSR[**XT**] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[**XT**].

**Special Registers Altered**

- **FX**
- **XX**

**VSR Data Layout for xvcvuxwdp**

<table>
<thead>
<tr>
<th>src = VSR[<strong>XB</strong>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[<strong>XT</strong>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**VSX Vector Convert and round Unsigned Integer Word to Single-Precision format**

**XX2-form**

**xvcvuxwsp**

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>168</th>
<th>8X</th>
</tr>
</thead>
</table>

**XT, XB**

- **XT** ← **TX** || **T**
- **XB** ← **BX** || **B**
- **ex_flag** ← 0b0

- do i=0 to 127 by 32
  - reset_xflags(!)
  - **v[0:inf]** ← ConvertUWtoFP(VSR[XB][i:i+31])
  - **result[i:i+31]** ← RoundToSP(RN,v)
  - if(xx_flag) then SetFX(XX)
  - **ex_flag** ← **ex_flag** | (XE & xx_flag)

- if( ex_flag = 0 ) then VSR[XT] ← result

Let **XT** be the value **TX** concatenated with **T**.
Let **XB** be the value **BX** concatenated with **B**.

For each vector element **i** from 0 to 127, do the following.
Let **src** be the unsigned integer in word element **i** of VSR[**XB**].

**src** is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field **RN** of the FPSCR.

The result is placed into word element **i** of VSR[**XT**] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[**XT**].

**Special Registers Altered**

- **FX**
- **XX**

**VSR Data Layout for xvcvuxwsp**

<table>
<thead>
<tr>
<th>src = VSR[<strong>XB</strong>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[<strong>XT</strong>]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
**VSX Vector Divide Double-Precision XX3-form**

### xvdivdp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>120</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>28</td>
</tr>
</tbody>
</table>

XT ← TX || T
XA ← AX || A
XB ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
reset_xflags()
srcl ← VSR[XA] {i:i+63}
srcl2 ← VSR[XB] {i:i+63}
v{0:inf} ← DivideDP(srch,src2)
result{i:i+63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag) then SetFX(VXIDI)
if(vxisi_flag) then SetFX(VXZDZ)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)
if(zx_flag) then SetFX(ZX)
if(xx_flag) then SetFX(XE)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxidi_flag)
ex_flag ← ex_flag | (VE & vxzdz_flag)
ex_flag ← ex_flag | (OE & ox_flag)
ex_flag ← ex_flag | (UE & ux_flag)
ex_flag ← ex_flag | (XE & xx_flag)
ex_flag ← ex_flag | (XX & zx_flag)
end
if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let srcl be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let srcl be the double-precision floating-point operand in doubleword element i of VSR[XB].

srcl is divided[1] by srcl2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 92.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

---

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significant left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significant was shifted.
Table 92. Actions for xvdvdp (element i)

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN vxidi_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1,sr2) vxzdz_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>vxzdz_flag ← 1</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN vxzdz_flag ← 1</td>
<td>v ← dQNaN vxzdz_flag ← 1</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN vxzdz_flag ← 1</td>
<td>v ← dQNaN vxzdz_flag ← 1</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1,sr2) vxzdz_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>vxzdz_flag ← 1</td>
<td>v ← -src1</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN vxzdz_flag ← 1</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>vxzdz_flag ← 1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:
- src1 The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
- src2 The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
- dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
- NZF Nonzero finite number.
- Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- D(x,y) Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
- Q(x) Return a QNaN with the payload of x.
- v The intermediate result having unbounded significand precision and unbounded exponent range.
### VSX Vector Divide Single-Precision XX3-form

The VSX Vector Divide Single-Precision XX3-form (`xdivsp XT,XA,XB`) performs floating-point division on single-precision elements of two vectors `XA` and `XB`, and stores the results in vector `XT`. Here's a breakdown of the operation:

1. **Input Vectors**:
   - `XT`: The result vector, where the elements are to be stored.
   - `XA`: The first operand vector.
   - `XB`: The second operand vector.

2. **Operation Details**:
   - For each vector element `i` from 0 to 127, the following steps are performed:
     - Let `src1` be the single-precision floating-point operand in word element `i` of `VSR[XA]`.
     - Let `src2` be the single-precision floating-point operand in word element `i` of `VSR[XB]`.
     - `src1` is divided by `src2`, producing a quotient with unbounded range and precision.
     - The quotient is normalized.
     - The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field `RN` of the FPSCR.

3. **Result Handling**:
   - The result is placed into word element `i` of `VSR[XT]` in single-precision format.

4. **Exception Handling**:
   - If a trap-enabled exception occurs in any element of the vector, no results are written to `VSR[XT].`

### Special Registers Altered

- `FX`, `OX`, `UX`, `ZX`, `XX`, `VXSNAN`, `VXIDI`, `VXZDZ`

### VSR Data Layout for `xdivsp`

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>src2 = VSR[XB]</th>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP SP SP SP</td>
<td>SP SP SP SP</td>
<td>SP SP SP SP</td>
</tr>
</tbody>
</table>

#### Notes:

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
Table 93. Actions for xdivsp (element i)

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td></td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← D(src1,src2)</td>
<td>v ← -Zero</td>
<td>v ← src2</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← -Zero</td>
<td>v ← D(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>

Explanation:
- src1: The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
- src2: The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
- dQNaN: Default quiet NaN (0x7FC0_0000).
- NZF: Nonzero finite number.
- Rezd: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
- D(x,y): Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
- Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- Q(x): Return a QNaN with the payload of x.
- v: The intermediate result having unbounded significand precision and unbounded exponent range.
### VSX Vector Multiply-Add Double-Precision XX3-form

#### xvmaddadp XT,XA,XB

<table>
<thead>
<tr>
<th>i</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>97</th>
<th>0b00X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0b00X</td>
</tr>
</tbody>
</table>

#### xvmaddmdp XT,XA,XB

<table>
<thead>
<tr>
<th>i</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>105</th>
<th>0b00X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0b00X</td>
</tr>
</tbody>
</table>

- **XT** ← TX || T
- **XA** ← AX || A
- **XB** ← BX || B
- **ex_flag** ← 0b0

```plaintext
do i=0 to 127 by 64
    reset_xflags()
    src1 ← VSR[XA]{i:i+63}
    src2 ← "xvmaddadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
    src3 ← "xvmaddadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
    v{0:inf} ← MultiplyAddDP(src1,src3,src2)
    result{i:i+63} ← RoundToDP(RN,v)
    if(vxsnan_flag) then SetFX(VXSNAN)
    if(vximz_flag) then SetFX(VXIMZ)
    if(vxisi_flag) then SetFX(VXISI)
    if(ox_flag) then SetFX(OX)
    if(ux_flag) then SetFX(UX)
    if(xx_flag) then SetFX(XX)
    ex_flag ← ex_flag | (VE & vxsnan_flag)
    ex_flag ← ex_flag | (VE & vximz_flag)
    ex_flag ← ex_flag | (VE & vxisi_flag)
    ex_flag ← ex_flag | (OE & ox_flag)
    ex_flag ← ex_flag | (UE & ux_flag)
    ex_flag ← ex_flag | (XE & xx_flag)
end
```

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX OX UX XX VXSNAN VXISI VXIMZ

For **xvmaddadp**, do the following.
- Let **src1** be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let **src2** be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let **src3** be the double-precision floating-point operand in doubleword element i of VSR[XB].

For **xvmaddmdp**, do the following.
- Let **src1** be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let **src2** be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let **src3** be the double-precision floating-point operand in doubleword element i of VSR[XB].

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>VSR Data Layout for xvmadd(alm)dp</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = VSR[XA]</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>src2 = xsmaddadp ? VSR[XT] : VSR[XB]</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>src3 = xsmaddadp ? VSR[XB] : VSR[XT]</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

0 64 127
<table>
<thead>
<tr>
<th>src</th>
<th>Multiply</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>p ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>Zero</td>
<td>p ← +Zero</td>
<td>v ← +Zero</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← -Zero</td>
<td>v ← -Zero</td>
</tr>
<tr>
<td>+Inf</td>
<td>p ← +Infinity</td>
<td>v ← +Infinity</td>
</tr>
<tr>
<td>NaN</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
</tr>
<tr>
<td>NaN</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
- **src2** For xvmadd dp, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
- **src3** For xvmadd dp, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
- **dQNaN** Default quiet NaN (Q=7F80_0000_0000_0000).
- **NZF** Nonzero finite number.
- **Rezd** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)** Return the normalized QNaN with the payload of x.
- **A(x,y)** Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
  
  **Note:** If x = -y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p** The intermediate product having unbounded range and precision.
- **v** The intermediate result having unbounded range and precision.

**Table 94. Actions for xvmadd(dp)**
VSX Vector Multiply-Add Single-Precision XX3-form

For \textit{xvmaddasp}, do the following.

- Let \textit{src1} be the single-precision floating-point operand in word element \textit{i} of VSR[\(\text{XA}\)].
- Let \textit{src2} be the single-precision floating-point operand in word element \textit{i} of VSR[\(\text{XB}\)].
- Let \textit{src3} be the single-precision floating-point operand in word element \textit{i} of VSR[\(\text{XT}\)].

\textit{src1} is multiplied\(^1\) by \textit{src3}, producing a product having unbounded range and precision.

\textit{src2} is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 1 of Table 95.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element \textit{i} of VSR[\(\text{XT}\)] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[\(\text{XT}\)].

**Special Registers Altered**

FX OX UX XX VXSNAN VXISI VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (\(G\), \(R\), and \(X\)) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvmadd(almsp)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = VSR[XA]</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src2 = xsmaddasp ? VSR[XT] : VSR[XB]</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src3 = xsmaddasp ? VSR[XB] : VSR[XT]</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>127</th>
</tr>
</thead>
</table>

Table 95. Actions for \texttt{xvmadd(alm)sp}

<table>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th>(-\infty)</th>
<th>(-\text{NZF})</th>
<th>(-\text{Zero})</th>
<th>(+\text{Zero})</th>
<th>(+\text{NZF})</th>
<th>(+\infty)</th>
<th>\text{QNaN}</th>
<th>\text{SNaN}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>p \leftarrow +\infty</td>
<td>p \leftarrow +\infty</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow -\infty</td>
<td>p \leftarrow +\text{src3}</td>
<td>p \leftarrow +\text{Q(src3)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>p \leftarrow +\infty</td>
<td>p \leftarrow +\text{M(src1,src3)}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{src3}</td>
<td>p \leftarrow +\text{Q(src3)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{src3}</td>
<td>p \leftarrow +\text{Q(src3)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>p \leftarrow -\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{src3}</td>
<td>p \leftarrow +\text{Q(src3)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>p \leftarrow -\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{src3}</td>
<td>p \leftarrow +\text{Q(src3)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
<tr>
<td>(+\infty)</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{Zero}</td>
<td>p \leftarrow +\text{dNaN} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{src3}</td>
<td>p \leftarrow +\text{Q(src3)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
<tr>
<td>\text{QNaN}</td>
<td>p \leftarrow +\text{src1}</td>
<td>p \leftarrow +\text{src1}</td>
<td>p \leftarrow +\text{src1}</td>
<td>p \leftarrow +\text{src1}</td>
<td>p \leftarrow +\text{src1}</td>
<td>p \leftarrow +\text{src1}</td>
<td>p \leftarrow +\text{src1}</td>
<td></td>
</tr>
<tr>
<td>\text{SNaN}</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td>p \leftarrow +\text{Q(src1)} \text{vximz_flag} \leftarrow 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 2: Add</th>
<th>(-\infty)</th>
<th>(-\text{NZF})</th>
<th>(-\text{Zero})</th>
<th>(+\text{Zero})</th>
<th>(+\text{NZF})</th>
<th>(+\infty)</th>
<th>\text{QNaN}</th>
<th>\text{SNaN}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>v \leftarrow +\infty</td>
<td>v \leftarrow +\infty</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>(-\text{NZF})</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>(-\text{Zero})</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>(+\text{Zero})</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>(+\text{NZF})</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>(+\text{Infinity})</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>\text{QNaN} &amp; \text{src1 is a NaN}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
<tr>
<td>\text{QNaN} &amp; \text{src1 not a NaN}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
<td>v \leftarrow +\text{Zero}</td>
</tr>
</tbody>
</table>

Explanation:

src1 The single-precision floating-point value in word element i of VSR[XT] (where i \in \{0,1,2,3\}).
src2 For \texttt{xvmaddasp}, the single-precision floating-point value in word element i of VSR[XT] (where i \in \{0,1,2,3\}). For \texttt{xvmaddmsp}, the single-precision floating-point value in word element i of VSR[XB] (where i \in \{0,1,2,3\}).
src3 For \texttt{xvmaddasp}, the single-precision floating-point value in word element i of VSR[XB] (where i \in \{0,1,2,3\}). For \texttt{xvmaddmsp}, the single-precision floating-point value in word element i of VSR[XT] (where i \in \{0,1,2,3\}).
d\text{QNaN} Default quiet NaN (0x7FEC_0000).
NZF Nonzero finite number.
Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x) Return a QNaN with the payload of x.
A(x,y) Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
Note: If x = -y, v is considered to be an exact-zero-difference result (Rezld).
M(x,y) Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
P The intermediate product having unbounded range and precision.
v The intermediate result having unbounded range and precision.
VSX Vector Maximum Double-Precision
XX3-form

\[ \text{xvmaxdp} \quad \text{XT,XA XB} \]

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>224</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do i=0 to 127 by 64} \\
\quad \text{reset_xflags()} \\
\quad \text{src1} \leftarrow \text{VSR[XA]}\{i:i+63\} \\
\quad \text{src2} \leftarrow \text{VSR[XB]}\{i:i+63\} \\
\quad \text{result}\{i:i+63\} \leftarrow \text{MaximumDP(src1,src2)} \\
\quad \text{if(vxsnan_flag) then SetFX(VXSNAN)} \\
\quad \text{ex_flag} \leftarrow \text{ex_flag | (VE & vxsnan_flag)} \\
\quad \text{end} \\
\quad \text{if( ex_flag = 0 ) then VSR[XT] ← result} \\
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[AX].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is greater than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 96.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**
FX VXSNAN

**VSX Data Layout for xvmaxdp**

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td>127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSR Data Layout for xvmaxdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = VSR[XA]</td>
</tr>
<tr>
<td>src2 = VSR[XB]</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>
## Table 96. Actions for `xvmaxdp`

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>–NZF</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>–Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(VXSNAN)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(M(src1,src2))</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(VXSNAN)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
<tr>
<td></td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
<td>T(VXSNAN)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (where \(i \in \{0,1\}\)).
- **src2** The double-precision floating-point value in doubleword element i of VSR[XT] (where \(i \in \{0,1\}\)).
- **NZF** Nonzero finite number.
- **Q(x)** Return a QNaN with the payload of x.
- **M(x,y)** Return the greater of floating-point value x and floating-point value y.
- **T(x)** The value x is placed in doubleword element i (i=0,1) of VSR[XT] in double-precision format.
- **FX**, **FR**, and **FI** are not modified.
- **fx(x)** If x is equal to 0, FX is set to 1. x is set to 1.
- **VXSNAN** Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.
VSX Vector Maximum Single-Precision
XX3-form

\[
x_{\text{vmaxsp}}(\text{XT,XA,XB}) \\
\]

<table>
<thead>
<tr>
<th>60</th>
<th>61</th>
<th>62</th>
<th>63</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>192</th>
<th>193</th>
<th>194</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{XT} \leftarrow \text{TX} \ | \ | \ T \\
\text{XA} \leftarrow \text{AX} \ | \ | \ A \\
\text{XB} \leftarrow \text{BX} \ | \ | \ B \\
\text{ex\_flag} \leftarrow 0b0 \\
\]

\[
do \text{ i = 0 to 127 by 32} \\
\text{reset\_xflags()} \\
\text{src1} \leftarrow \text{VSR}[\text{XA}]_{\text{i:i+31}} \\
\text{src2} \leftarrow \text{VSR}[\text{XB}]_{\text{i:i+31}} \\
\text{result}_{\text{i:i+63}} \leftarrow \text{MaximumSP}(\text{src1,src2}) \\
\text{if} (\text{vxsnan\_flag}) \text{ then SetFX(VXSNAN)} \\
\text{ex\_flag} \leftarrow \text{ex\_flag} \ | \ (\text{VE} \ & \ \text{vxsnan\_flag}) \\
\text{end} \\
\text{if( ex\_flag } = 0 \text{ ) then VSR}[\text{XT}] \leftarrow \text{result} \\
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is greater than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The maximum of +0 and −0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 97.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX  VXSNAN

**VSR Data Layout for xvmaxsp**

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>
### Table 97. Actions for `xvmaxsp`

<table>
<thead>
<tr>
<th>Table 97. Actions for <code>xvmaxsp</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>src2</strong></td>
</tr>
<tr>
<td><strong>–Infinity</strong></td>
</tr>
<tr>
<td><strong>–NZF</strong></td>
</tr>
<tr>
<td><strong>–Zero</strong></td>
</tr>
<tr>
<td><strong>+Zero</strong></td>
</tr>
<tr>
<td><strong>+NZF</strong></td>
</tr>
<tr>
<td><strong>+Infinity</strong></td>
</tr>
<tr>
<td><strong>QNaN</strong></td>
</tr>
<tr>
<td><strong>SNaN</strong></td>
</tr>
</tbody>
</table>

#### Explanation:

- **src1**: The single-precision floating-point value in word element `i` of VSR[XA] (where `i ∈ {0,1,2,3}`).
- **src2**: The single-precision floating-point value in word element `i` of VSR[XT] (where `i ∈ {0,1,2,3}`).
- **NZF**: Nonzero finite number.
- **Q(x)**: Return a QNaN with the payload of `x`.
- **M(x,y)**: Return the greater of floating-point value `x` and floating-point value `y`.
- **T(x)**: The value `x` is placed in word element `i` (where `i ∈ {0,1,2,3}`) of VSR[XT] in single-precision format.
- **FX**, **FR** and **FI** are not modified.
- **fx(x)**: If `x` is equal to 0, FX is set to 1. `x` is set to 1.
- **VXSNAN**: Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.
VSX Vector Minimum Double-Precision
XX3-form

xvmindp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>232</th>
<th>0b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>23</td>
</tr>
</tbody>
</table>

% $XT \leftarrow TX \| T$

% $XA \leftarrow AX \| A$

% $XB \leftarrow BX \| B$

% $\text{ex\_flag} \leftarrow 0b0$

\[
do\ i=0\ \text{to}\ 127\ \text{by}\ 64
\]
\[
\text{reset\_xflags()}
\]
\[
\text{src1} \leftarrow \text{VSR}[\text{XA}]_{i:i+63}
\]
\[
\text{src2} \leftarrow \text{VSR}[\text{XB}]_{i:i+63}
\]
\[
\text{result}_{i:i+63} \leftarrow \text{MinimumDP}(\text{src1}, \text{src2})
\]
\[
\text{if(vxsnan\_flag) then SetFX(VXSNAN)}
\]
\[
\text{ex\_flag} \leftarrow \text{ex\_flag} \text{ | (VE \& vxsnan\_flag)}
\]
\[\text{end}\]
\[
\text{if( ex\_flag = 0 ) then VSR[XT] \leftarrow \text{result}}
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNan when VE=0 is that SNan converted to a QNaN.

See Table 98.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX VXSNAN
### Table 98. Actions for `xvmindp`

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td>-NZF</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)(VXSNAN)</td>
</tr>
<tr>
<td>-Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)(VXSNAN)</td>
</tr>
<tr>
<td>+Zero</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)(VXSNAN)</td>
</tr>
<tr>
<td>+NZF</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)(VXSNAN)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)(VXSNAN)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)(VXSNAN)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
</tr>
</tbody>
</table>

**Explanation:**

src1 The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).  
src2 The double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).  
NZF Nonzero finite number.  
Q(x) Return a QNaN with the payload of x.  
M(x,y) Return the lesser of floating-point value x and floating-point value y.  
T(x) The value x is placed in doubleword element i (i=(0,1)) of VSR[XT] in double-precision format.  
FPRF, FR and FI are not modified.  
f(x) If x is equal to 0, FX is set to 1. x is set to 1.  
VXSNAN Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.
**VSX Vector Minimum Single-Precision XX3-form**

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is less than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 99.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX  VXSNAN
### Table 99. Actions for `xvminsp`

<table>
<thead>
<tr>
<th><code>src2</code></th>
<th><code>–Infinity</code></th>
<th><code>–NZF</code></th>
<th><code>–Zero</code></th>
<th><code>+Zero</code></th>
<th><code>+NZF</code></th>
<th><code>+Infinity</code></th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>–Infinity</code></td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(Q(src2))</td>
<td>T(Q(src2))</td>
</tr>
<tr>
<td><code>–NZF</code></td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)</td>
</tr>
<tr>
<td><code>–Zero</code></td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(x)</td>
</tr>
<tr>
<td><code>+Zero</code></td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(x)</td>
</tr>
<tr>
<td><code>+NZF</code></td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(M(src1,src2))</td>
<td>T(src1)</td>
<td>T(src1)</td>
<td>T(x)</td>
</tr>
<tr>
<td><code>+Infinity</code></td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(src2)</td>
<td>T(x)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(x)</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(Q(src1))</td>
<td>T(x)</td>
</tr>
</tbody>
</table>

#### Explanation:
- **src1** The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
- **src2** The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
- **NZF** Nonzero finite number.
- **Q(x)** Return a QNaN with the payload of x.
- **M(x,y)** Return the lesser of floating-point value x and floating-point value y.
- **T(x)** The value x is placed in word element i (i ∈ {0,1,2,3}) of VSR[XT] in single-precision format.
- **FPRF, FR and FI are not modified.**
- **fx(x)** If x is equal to 0, FX is set to 1, x is set to 1.
- **VXSNAN** Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.
**VSX Vector Multiply-Subtract**

**Double-Precision XX3-form**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{xvmsubadp}</td>
<td>XT,XA,XB</td>
</tr>
</tbody>
</table>

| XT | ← TX || T |
| XA | ← AX || A |
| XB | ← BX || B |
| ex_flag | ← 0b0 |

For \text{xvmsubadp}, do the following.

- Let \text{src1} be the double-precision floating-point operand in doubleword element \(i\) of VSR[ZA].
- Let \text{src2} be the double-precision floating-point operand in doubleword element \(i\) of VSR[ZB].
- Let \text{src3} be the double-precision floating-point operand in doubleword element \(i\) of VSR[XZ].

\text{src1} is multiplied\(^1\) by \text{src3}, producing a product having unbounded range and precision.

\text{src2} is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 1 of Table 100.

See part 2 of Table 100.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element \(i\) of VSR[XZ] in double-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XZ].

**Special Registers Altered**

- FX
- OX
- XX
- VXSNAN
- VXISI
- VXIMZ

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
VSR Data Layout for `xvmsub(alm)dp`

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th>DP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt = VSR[XT]</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>
Table 100. Actions for `xvmsub(aim)dp`

<table>
<thead>
<tr>
<th>Part 1: Multiply</th>
<th>src3</th>
<th>src1</th>
<th>src2</th>
<th>src0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-Infinity</td>
<td>-NZF</td>
<td>-Zero</td>
<td>+Zero</td>
</tr>
<tr>
<td>-Infinity</td>
<td>p ← -Infinity</td>
<td>p ← -Infinity</td>
<td>p ← +NZF</td>
<td>p ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>p ← -Infinity</td>
<td>p ← M(src1, src3)</td>
<td>p ← +Zero</td>
<td>p ← -Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>p ← +Infinity</td>
<td>p ← +NZF</td>
<td>p ← +Zero</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← -Infinity</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← -Infinity</td>
<td>p ← -Zero</td>
<td>p ← +Zero</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← -Infinity</td>
<td>p ← +Infinity</td>
<td>p ← -Zero</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
<td>p ← Q(src1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 2: Subtract</th>
<th>src2</th>
<th>src1</th>
<th>src0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-Infinity</td>
<td>-NZF</td>
<td>-Zero</td>
</tr>
<tr>
<td>-Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
</tr>
<tr>
<td>QNaN &amp; src1 a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1:** The double-precision floating-point value in doubleword element i of `VSR[XT]` (where i ∈ (0, 1)).
- **src2:** For `xvmsubadp`, the double-precision floating-point value in doubleword element i of `VSR[XA]` (where i ∈ (0, 1)).
- **src3:** For `xvmsubadp`, the double-precision floating-point value in doubleword element i of `VSR[XB]` (where i ∈ (0, 1)).
- **dQNaN:** Default quiet NaN (`QNaN00_0000_0000_0000`).
- **NZF:** Nonzero finite number.
- **Rezd:** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number operands.
- **Q(x):** Return a QNaN with the payload of x.
- **S(x, y):** Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
  - **Note:** If x = y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x, y):** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p:** The intermediate product having unbounded range and precision.
- **v:** The intermediate result having unbounded range and precision.
VSX Vector Multiply-Subtract Single-Precision XX3-form

For **xvmsubasp**, do the following.
- Let $src_1$ be the single-precision floating-point operand in word element $i$ of VSR[XA].
- Let $src_2$ be the single-precision floating-point operand in word element $i$ of VSR[XB].
- Let $src_3$ be the single-precision floating-point operand in word element $i$ of VSR[XT].

For **xvmsubmsp**, do the following.
- Let $src_1$ be the single-precision floating-point operand in word element $i$ of VSR[XA].
- Let $src_2$ be the single-precision floating-point operand in word element $i$ of VSR[XB].
- Let $src_3$ be the single-precision floating-point operand in word element $i$ of VSR[XT].

Let $XT$ be the value $TX$ concatenated with $T$.
Let $XA$ be the value $AX$ concatenated with $A$.
Let $XB$ be the value $BX$ concatenated with $B$.

For each vector element $i$ from 0 to 3, do the following.

For **xvmsubasp**, do the following.
- Let $src_1$ be the single-precision floating-point operand in word element $i$ of VSR[XA].
- Let $src_2$ be the single-precision floating-point operand in word element $i$ of VSR[XT].
- Let $src_3$ be the single-precision floating-point operand in word element $i$ of VSR[XB].

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits ($G$, $R$, and $X$) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = <em>xvmsubasp</em> ? VSR[XT] : VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src3 = <em>xvmsubasp</em> ? VSR[XB] : VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

0 32 64 96 127
### Table 101. Actions for `xvmsub(al)sp`

<table>
<thead>
<tr>
<th>src1</th>
<th>src2</th>
<th>src3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$p$</td>
<td>$p$</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$p$</td>
<td>$p$</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1**: The single-precision floating-point value in word element $i$ of VSR[XT] (where $i \in \{0, 1, 2, 3\}$).
- **src2**: For `xvmsubasp`, the single-precision floating-point value in word element $i$ of VSR[XT] (where $i \in \{0, 1, 2, 3\}$).
- **src3**: For `xvmsubasp`, the single-precision floating-point value in word element $i$ of VSR[XB] (where $i \in \{0, 1, 2, 3\}$).
- **dQNaN**: Default quiet NaN (0x7FC0_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of $x$.
- **S(x,y)**: Return the normalized sum of floating-point value $x$ and negated floating-point value $y$, having unbounded range and precision.
  - **Note**: If $x = y$, $v$ is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)**: Return the normalized product of floating-point value $x$ and floating-point value $y$, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
VSX Vector Multiply Double-Precision
XX3-form

xvmuldp   XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>←</td>
<td>TX</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>←</td>
<td>AX</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>←</td>
<td>BX</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>←</td>
<td>0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 64
reset_xflags()
  src1 ← VSR[XA]{i:i+63}
  src3 ← VSR[XB]{i:i+63}
  v{0:inf} ← MultiplyDP(src1,src3)
  result{i:i+63} ← RoundToDP(RN,v)
  if(vxsnan_flag) then SetFX(VXSNAN)
  if(vximz_flag)  then SetFX(VXIMZ)
  if(ox_flag)     then SetFX(OX)
  if(ux_flag)     then SetFX(UX)
  if(xx_flag)     then SetFX(XX)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
  ex_flag ← ex_flag | (VE & vximz_flag)
  ex_flag ← ex_flag | (OE & ox_flag)
  ex_flag ← ex_flag | (UE & ux_flag)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point
operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point
operand in doubleword element i of VSR[XB].

src1 is multiplied\(^1\) by src2, producing a product
having unbounded range and precision.

The product is normalized\(^2\).

See Table 102.

The intermediate result is rounded to
double-precision using the rounding mode
specified by the Floating-Point Rounding Control
field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result
Handling,” on page 402.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Table 102. Actions for xvmuldp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← +Infinity v ← +Infinity v ← dQNaN v ← dQNaN v ← –Infinity v ← –Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity v ← M(src1, src2) v ← –Zero v ← +Infinity v ← M(src1, src2) v ← –Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +QNaN v ← +QNaN v ← +Zero v ← –Zero v ← +Zero v ← dQNaN v ← vximz_flag ← 1 v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +QNaN v ← +QNaN v ← +Zero v ← –Zero v ← +Zero v ← dQNaN v ← vximz_flag ← 1 v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← –Infinity v ← M(src1, src2) v ← –Zero v ← +Infinity v ← M(src1, src2) v ← –Infinity v ← src2 v ← Q(src2) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1 v ← src1 v ← src1 v ← src1 v ← src1 v ← src1 v ← src1 v ← src1 v ← src1 vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1 v ← Q(src1) vxsnan_flag ← 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Explanation:
- **src1** The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0, 1}).
- **src2** The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0, 1}).
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **M(x, y)** Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.
VSX Vector Multiply Single-Precision

**XX3-form**

### xvmulsp

XT, XA, XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>80</th>
<th>A8</th>
<th>B8</th>
</tr>
</thead>
</table>

XT ← TX || T
XA ← AX || A
XB ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
  reset_xflags()
  src1 ← VSR[XA]{i:i+31}
  src3 ← VSR[XB]{i:i+31}
  v{0:inf} ← MultiplySP(src1,src3)
  result{i:i+31} ← RoundToSP(RN,v)
  if(vxsnan_flag) then SetFX(VXSNAN)
  if(vximz_flag)  then SetFX(VXIMZ)
  if(ox_flag)     then SetFX(OX)
  if(ux_flag)     then SetFX(UX)
  if(xx_flag)     then SetFX(XX)
  ex_flag ← ex_flag | (VE & vxsnan_flag)
  ex_flag ← ex_flag | (VE & vximz_flag)
  ex_flag ← ex_flag | (OE & ox_flag)
  ex_flag ← ex_flag | (UE & ux_flag)
  ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is multiplied\(^1\) by src2, producing a product having unbounded range and precision.

The product is normalized\(^2\).

See Table 103.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### Explanation:

- **src1**: The single-precision floating-point value in word element `i` of VSR[XA] (where `i` ∈ {0,1,2,3}).
- **src2**: The single-precision floating-point value in word element `i` of VSR[XB] (where `i` ∈ {0,1,2,3}).
- **dQNaN**: Default quiet NaN (0x7FC0_0000).
- **NZF**: Nonzero finite number.
- **M(x,y)**: Return the normalized product of floating-point value `x` and floating-point value `y`, having unbounded range and precision.
- **Q(x)**: Return a QNaN with the payload of `x`.
- **v**: The intermediate result having unbounded significand precision and unbounded exponent range.

### Table 103. Actions for xvmulsp

<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← -Infinity</td>
<td>v ← M(src1,src2)</td>
<td>v ← -Zero</td>
<td>v ← -Zero</td>
<td>v ← M(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← -Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← +Zero</td>
<td>v ← dQNaN</td>
<td>v ← +Zero</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← -Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN</td>
<td>v ← dQNaN</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2)</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
<td>v ← Q(src1)</td>
</tr>
</tbody>
</table>
VSX Vector Negative Absolute Value
Double-Precision XX2-form

\[ \text{xvnabsdp } XT, XB \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 0 )</th>
<th>( 60 )</th>
<th>( T )</th>
<th>( / / / )</th>
<th>( B )</th>
<th>( 489 )</th>
<th>( \text{SBX} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( XT )</td>
<td>( TX )</td>
<td></td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( XB )</td>
<td>( BX )</td>
<td></td>
<td>( B )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 64 \\
\quad \text{VSR[XT]}\{i:i+63\} \leftarrow 0b1 || \text{VSR[XB]}\{i+1:i+63\} \\
\text{end}
\]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

For each vector element \( i \) from 0 to 1, do the following.
The contents of doubleword element \( i \) of VSR[XB], with bit 0 set to 1, is placed into doubleword element \( i \) of VSR[XT].

Special Registers Altered
None

VSX Vector Negative Absolute Value
Single-Precision XX2-form

\[ \text{xvnabssp } XT, XB \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 0 )</th>
<th>( 60 )</th>
<th>( T )</th>
<th>( / / / )</th>
<th>( B )</th>
<th>( 425 )</th>
<th>( \text{SBX} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( XT )</td>
<td>( TX )</td>
<td></td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( XB )</td>
<td>( BX )</td>
<td></td>
<td>( B )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\quad \text{VSR[XT]}\{i:i+31\} \leftarrow 0b1 || \text{VSR[XB]}\{i+1:i+31\} \\
\text{end}
\]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

For each vector element \( i \) from 0 to 3, do the following.
The contents of word element \( i \) of VSR[XB], with bit 0 set to 1, is placed into word element \( i \) of VSR[XT].

Special Registers Altered
None
VSX Vector Negate Double-Precision
XX2-form

\[
\text{xvnegdp XT,XB}
\]

\[
\begin{array}{cccccc}
60 & T & /// & B & 505 & BX\\
0 & 6 & 11 & 16 & 21 & 511
\end{array}
\]

\[
\begin{align*}
\text{XT} & \leftarrow TX || T \\
\text{XB} & \leftarrow BX || B \\
\text{do } i=0 \text{ to } 127 \text{ by } 64 \\
\text{VSR[XT][i:i+63]} & \leftarrow \sim\text{VSR[XB][i]} || \text{VSR[XB][i+1:i+63]}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 complemented, is placed into doubleword element i of VSR[XT].

Special Registers Altered
None

VSR Data Layout for xvnegdp
\[
\begin{array}{c|c|c|c|c}
\text{src} = \text{VSR[XB]}\\
\hline
\text{DP} & \text{DP}\\
\text{tgt} = \text{VSR[XT]}\\
\hline
0 & 64 & 127
\end{array}
\]

VSX Vector Negate Single-Precision
XX2-form

\[
\text{xvnegsp XT,XB}
\]

\[
\begin{array}{cccccc}
60 & T & /// & B & 441 & BX\\
0 & 6 & 11 & 16 & 21 & 511
\end{array}
\]

\[
\begin{align*}
\text{XT} & \leftarrow TX || T \\
\text{XB} & \leftarrow BX || B \\
\text{do } i=0 \text{ to } 127 \text{ by } 32 \\
\text{VSR[XT][i:i+31]} & \leftarrow \sim\text{VSR[XB][i]} || \text{VSR[XB][i+1:i+31]}
\end{align*}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 complemented, is placed into word element i of VSR[XT].

Special Registers Altered
None

VSR Data Layout for xvnegsp
\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{src} = \text{VSR[XB]}\\
\hline
\text{SP} & \text{SP} & \text{SP} & \text{SP}\\
\text{tgt} = \text{VSR[XT]}\\
\hline
0 & 32 & 64 & 96 & 127
\end{array}
\]
VSX Vector Negative Multiply-Add
Double-Precision XX3-form

For `xvnmmaddp`, do the following.
- Let `src1` be the double-precision floating-point operand in doubleword element `i` of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element `i` of VSR[XB].
- Let `src3` be the double-precision floating-point operand in doubleword element `i` of VSR[XT].

`src1` is multiplied\(^1\) by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 104.

`src2` is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 104.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element `i` of VSR[XT] in double-precision format.

See Table 105, “Vector Floating-Point Final Result with Negation,” on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
```
  FX OX UX XX VXSNaN VXISI VXIMZ
```

For `xvnmmaddmdp`, do the following.
- Let `src1` be the double-precision floating-point operand in doubleword element `i` of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element `i` of VSR[XB].
- Let `src3` be the double-precision floating-point operand in doubleword element `i` of VSR[XT].

`src1` is multiplied\(^1\) by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 104.

`src2` is added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 104.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element `i` of VSR[XT] in double-precision format.

See Table 105, “Vector Floating-Point Final Result with Negation,” on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
```
  FX OX UX XX VXSNaN VXISI VXIMZ
```

Floating-point multiplication is based on exponent addition and multiplication of the significands.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

---

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for `xvmadd(al)m`dp

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = <code>xsmaddadp</code> ? VSR[XT] : VSR[XB]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src3 = <code>xsmaddadp</code> ? VSR[XB] : VSR[XT]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

0 64 127
Table 104. Actions for `vxnmmadd(a|m)dp`

<table>
<thead>
<tr>
<th>src3</th>
<th>Multiply</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>( p \leftarrow \text{\text{-Infinity}} ) ( p \leftarrow dONaN )</td>
<td>( v \leftarrow \text{\text{-Infinity}} ) ( v \leftarrow dONaN )</td>
</tr>
<tr>
<td>-NZF</td>
<td>( p \leftarrow \text{\text{-NZF}} ) ( p \leftarrow \text{\text{-NZF}} )</td>
<td>( v \leftarrow \text{\text{-NZF}} ) ( v \leftarrow \text{\text{-NZF}} )</td>
</tr>
<tr>
<td>-Zero</td>
<td>( p \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 ) ( p \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 )</td>
<td>( v \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 ) ( v \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 )</td>
</tr>
<tr>
<td>+Zero</td>
<td>( p \leftarrow \text{\text{+Zero}} ) ( p \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 )</td>
<td>( v \leftarrow \text{\text{+Zero}} ) ( v \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 )</td>
</tr>
<tr>
<td>+NZF</td>
<td>( p \leftarrow \text{\text{+NZF}} ) ( p \leftarrow \text{\text{+NZF}} )</td>
<td>( v \leftarrow \text{\text{+NZF}} ) ( v \leftarrow \text{\text{+NZF}} )</td>
</tr>
<tr>
<td>+Infinity</td>
<td>( p \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 )</td>
<td>( v \leftarrow dONaN ) ( \text{\text{vximz_flag}} \leftarrow 1 )</td>
</tr>
<tr>
<td>QNaN</td>
<td>( p \leftarrow \text{\text{Q(src1)}} ) ( \text{\text{vxsnan_flag}} \leftarrow 1 )</td>
<td>( v \leftarrow \text{\text{Q(src1)}} ) ( \text{\text{vxsnan_flag}} \leftarrow 1 )</td>
</tr>
<tr>
<td>SNaN</td>
<td>( p \leftarrow \text{\text{Q(src1)}} ) ( \text{\text{vxsnan_flag}} \leftarrow 1 )</td>
<td>( v \leftarrow \text{\text{Q(src1)}} ) ( \text{\text{vxsnan_flag}} \leftarrow 1 )</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1:** The double-precision floating-point value in doubleword element \( i \) of VSR[XA] (where \( i \in (0,1) \)).
- **src2:** For `vxnmmadddp`, the double-precision floating-point value in doubleword element \( i \) of VSR[XT] (where \( i \in (0,1) \)).
- **src3:** For `vxnmmadddp`, the double-precision floating-point value in doubleword element \( i \) of VSR[XB] (where \( i \in (0,1) \)).
- **dONaN:** Default quiet NaN \((0x07F8_0000_0000_0000)\).
- **NZF:** Nonzero finite number.
- **Rezd:** Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number operand.
- **Q(x):** Return a QNaN with the payload of \( x \).
- **A(x,y):** Return the normalized sum of floating-point value \( x \) and floating-point value \( y \), having unbounded range and precision.
- **M(x,y):** Return the product of floating-point value \( x \) and floating-point value \( y \), having unbounded range and precision.
- **p:** The intermediate product having unbounded range and precision.
- **v:** The intermediate result having unbounded range and precision.
<table>
<thead>
<tr>
<th>Case</th>
<th>VE</th>
<th>OE</th>
<th>UE</th>
<th>ZE</th>
<th>XE</th>
<th>vxsnan_flag</th>
<th>vximz_flag</th>
<th>vxisi_flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Normal</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td>Overflow</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Returned Results and Status Setting**

- Ret(\(r\))
- T(\(r\)), fx(VXISI)
- T(\(r\)), fx(VXIMZ)
- T(\(r\)), fx(VXSNAN)
- T(\(r\)), fx(VXSNAN), fx(VXIMZ)
- T(\(r\)), fx(VXSNAN), error()
- fx(VXIMZ), error()
- fx(VXSNAN), error()
- fx(VXSNAN), fx(VXIMZ), error()
- fx(VXISI), error()
- fx(VXIMZ), error()
- fx(VXSNAN), error()
- fx(VXSNAN), fx(VXIMZ), error()
- fx(VXISI), error()
- fx(VXIMZ), error()
- fx(VXSNAN), error()
- fx(VXSNAN), fx(VXIMZ), error()

**Explanation:**

- The results do not depend on this condition.
- FX is set to 1 if \(x=0\). \(x\) is set to 1.
- The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, unbounded exponent range.
- The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, significand rounded to the target precision, bounded exponent range.
- The precise intermediate result defined in the instruction having unbounded significand precision, unbounded exponent range.
- Floating-Point Fraction Inexact status flag, FPSCR\_IF. This status flag is nonsticky.
- Floating-Point Fraction Rounded status flag, FPSCR\_FR.
- Floating-Point Overflow Exception status flag, FPSCR\_OX.
- The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
- The value \(x\) is is negated by complementing the sign bit of \(x\).
- The value \(x\) is placed in element i of VSR[\(XT\)] in the target precision format (where i \(\in\{0,1\}\) for results with 64-bit elements, and i \(\in\{0,1,3,4\}\) for results with 32-bit elements).
- Floating-Point Underflow Exception status flag, FPSCR\_UX.
- Floating-Point Invalid Operation Exception (SNAN) status flag, FPSCR\_VXSNAN.
- Floating-Point Invalid Operation Exception (Infinity \(\times\) Zero) status flag, FPSCR\_VXIMZ.
- Floating-Point Invalid Operation Exception (Infinity \(\div\) Infinity) status flag, FPSCR\_VXISI.
- Floating-Point Inexact Exception status flag, FPSCR\_XX. The flag is a sticky version of FPSCR\_F. When FPSCR\_F is set to a new value, the new value of FPSCR\_XX is set to the result of ORing the old value of FPSCR\_XX with the new value of FPSCR\_F.

**Table 105. Vector Floating-Point Final Result with Negation**
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX] 551

Table 105. Vector Floating-Point Final Result with Negation (Continued)
**VSX Vector Negative Multiply-Add**

**Single-Precision XX3-form**

For **vxnmadmsp**, do the following.
- Let **src1** be the single-precision floating-point operand in word element i of VSR[XA].
- Let **src2** be the single-precision floating-point operand in word element i of VSR[XB].
- Let **src3** be the single-precision floating-point operand in word element i of VSR[XT].

**src1** is multiplied[1] by **src3**, producing a product having unbounded range and precision.

See part 1 of Table 106.

**src2** is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 106.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 105, “Vector Floating-Point Final Result with Negation,” on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX OX UX XX VXSNAN VXISI VXIMZ

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

For **vxnmadasp**, do the following.
- Let **src1** be the single-precision floating-point operand in word element i of VSR[XA].
- Let **src2** be the single-precision floating-point operand in word element i of VSR[XT].
- Let **src3** be the single-precision floating-point operand in word element i of VSR[XB].

Let **ex_flag** ← 0b0

do i=0 to 127 by 32
reset_xflags();
src1 ← VSR[XA]|i:i+31|
v|0:inf| ← MultiplyAddSP(src1,src3,src2)
result{i:i+31} ← NegateSP(RoundToSP(RN,v))
if(v|vexsnan_flag) then SetFX(VXSNAN)
if(v|vxisi_flag)  then SetFX(VXISI)
if(v|ox_flag)     then SetFX(OX)
if(v|ux_flag)     then SetFX(UX)
if(v|xx_flag)     then SetFX(XX)
ex_flag ← ex_flag | (VE & vexsnan_flag)
ex_flag ← ex_flag | (VE & vxisi_flag)
ex_flag ← ex_flag | (VE & ox_flag)
ex_flag ← ex_flag | (VE & ux_flag)
ex_flag ← ex_flag | (VE & xx_flag)
ex_flag ← ex_flag | (VE & ox_flag)
ex_flag ← ex_flag | (VE & ux_flag)
ex_flag ← ex_flag | (VE & xx_flag)
end
if( ex_flag = 0 ) then VSR[XT] ← result

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for xvnadd(alm)sp

- **src1 = VSR[XA]**
  - SP | SP | SP | SP

- **src2 = \texttt{xsmaddadp} ? \texttt{VSR[XT]} : \texttt{VSR[XB]}**
  - SP | SP | SP | SP

- **src3 = \texttt{xsmaddadp} ? \texttt{VSR[XB]} : \texttt{VSR[XT]}**
  - SP | SP | SP | SP

- **tgt = VSR[XT]**
  - SP | SP | SP | SP

| 0 | 32 | 64 | 96 | 127 |
### Table 106. Actions for \texttt{xvnmadd(al)m}sp

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>+Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p ← dQNaN</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>p ← src1</td>
<td>p ← dQNaN</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← –Zero</td>
<td>M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← –Infinity</td>
<td>+NZF</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
<td>Q(src1)</td>
</tr>
</tbody>
</table>

**Part 1: Multiply**

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>+Infinity</td>
<td>p ← dQNaN</td>
<td>p ← dQNaN</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>–Zero</td>
<td>p ← +Zero</td>
<td>p ← src1</td>
<td>p ← +Zero</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← –Infinity</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← –Infinity</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← –Infinity</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>p ← –Infinity</td>
<td>p ← src1</td>
<td>p ← –Infinity</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
</tbody>
</table>

**Part 2: Add**

<table>
<thead>
<tr>
<th>src2</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← +Infinity</td>
<td>p ← dQNaN</td>
<td>+Infinity</td>
<td>p ← dQNaN</td>
<td>p ← dQNaN</td>
</tr>
<tr>
<td>–NZF</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
<td>p ← src1</td>
<td>p ← +Infinity</td>
</tr>
<tr>
<td>–Zero</td>
<td>p ← –Zero</td>
<td>p ← src1</td>
<td>p ← –Zero</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>+Zero</td>
<td>p ← +Zero</td>
<td>p ← src1</td>
<td>p ← +Zero</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>+NZF</td>
<td>p ← +Zero</td>
<td>p ← src1</td>
<td>p ← +Zero</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>p ← +Zero</td>
<td>p ← src1</td>
<td>p ← +Zero</td>
<td>src1</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>p ← src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
<td>src1</td>
</tr>
</tbody>
</table>

**Explanation:**

- **src1** The single-precision floating-point value in word element i of VSR[XA] (where i \( \in \{0,1,2,3\} \)).
- **src2** For xvnmaddasp, the single-precision floating-point value in word element i of VSR[XT] (where i \( \in \{0,1,2,3\} \)).
- **src3** For xvnmaddasp, the single-precision floating-point value in word element i of VSR[XB] (where i \( \in \{0,1,2,3\} \)).
- For \texttt{xvnmaddmsp}, the single-precision floating-point value in word element i of VSR[XBT] (where i \( \in \{0,1,2,3\} \)).
- dQNaN Default quiet NaN (0x7FC0_0000).
- NZF Nonzero finite number.
- Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- Q(x) Return a QNaN with the payload of x.
- A(x,y) Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
  - Note: If x = –y, v is considered to be an exact-zero-difference result (Rezd).
- M(x,y) Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- p The intermediate product having unbounded range and precision.
- v The intermediate result having unbounded range and precision.
VSX Vector Negative Multiply-Subtract
Double-Precision XX3-form

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{xnmsubadp} & \text{XT,XA,XB} \\
\hline
60 & B & A & B & 241 & \text{VPXX} \\
\hline
0 & 11 & 16 & 21 & \text{VXSNAN, VXISI, VXIMZ} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{xnmsubmdp} & \text{XT,XA,XB} \\
\hline
60 & B & A & B & 249 & \text{VPXX} \\
\hline
0 & 11 & 16 & 21 & \text{VXSNAN, VXISI, VXIMZ} \\
\hline
\end{array}
\]

\[
\begin{align*}
\text{XT} & \leftarrow \text{TX} \mid T \\
\text{XA} & \leftarrow \text{AX} \mid A \\
\text{XB} & \leftarrow \text{BX} \mid B \\
\text{ex\_flag} & \leftarrow 0b0
\end{align*}
\]

do i=0 to 127 by 64
reset_xflags()
src1 \leftarrow \text{VSR[XA]}_{i:i+63}
src2 \leftarrow "xnmsubadp" ? \text{VSR[XT]}_{i:i+63} : \text{VSR[XB]}_{i:i+63}
src3 \leftarrow "xnmsubadp" ? \text{VSR[XT]}_{i:i+63} : \text{VSR[XB]}_{i:i+63}
v(\text{Vxsnan\_flag}) \leftarrow \text{MultiplyAddDP(src1,src3,NegateDP(src2))}
result_{i:i+63} \leftarrow \text{NegateDP(RoundToDP(RN,v))}
\text{if(vxsnan\_flag)} \text{ then SetFX(VXSNAN)}
\text{if(vximz\_flag)} \text{ then SetFX(VXIMZ)}
\text{if(vxisi\_flag)} \text{ then SetFX(VXISI)}
\text{if(ox\_flag)} \text{ then SetFX(OX)}
\text{if(ux\_flag)} \text{ then SetFX(UX)}
\text{if(xx\_flag)} \text{ then SetFX(XX)}
ex\_flag \leftarrow ex\_flag | (\text{VE & vxsnan\_flag})
ex\_flag \leftarrow ex\_flag | (\text{VE & vximz\_flag})
ex\_flag \leftarrow ex\_flag | (\text{VE & vxisi\_flag})
ex\_flag \leftarrow ex\_flag | (\text{OE & ox\_flag})
ex\_flag \leftarrow ex\_flag | (\text{UE & ux\_flag})
ex\_flag \leftarrow ex\_flag | (\text{XE & xx\_flag})
\text{end}
\text{if(ex\_flag = 0)} \text{ then VSR[XT] \leftarrow result}
\]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

For \text{xnmsubadp}, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 107.

src2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 107.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 105, “Vector Floating-Point Final Result with Negation,” on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

<table>
<thead>
<tr>
<th>FX</th>
<th>OX</th>
<th>UX</th>
<th>XX</th>
<th>VXSNAN</th>
<th>VXISI</th>
<th>VXIMZ</th>
</tr>
</thead>
</table>

\(^1\) Floating-point multiplication is based on exponent addition and multiplication of the significands.
\(^2\) Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
\(^3\) Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
**VSR Data Layout for xvnmsub(alm)dp**

src1 = VSR[XA]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>

src2 = \textit{xvnmsubadp} \ ? VSR[XT] : VSR[XB]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>

src3 = \textit{xvnmsubadp} \ ? VSR[XB] : VSR[XB]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
</tr>
</tbody>
</table>

tgt = VSR[XT]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

0       64    127
Table 107. Actions for xvnmsub(a|m)dp

<table>
<thead>
<tr>
<th>src3</th>
<th>(-\infty)</th>
<th>(-NZF)</th>
<th>(-Zero)</th>
<th>(+Zero)</th>
<th>(+NZF)</th>
<th>(+\infty)</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>p ← dQNaN vxinz_flag ← 1</td>
<td>p ← dQNaN vxinz_flag ← 1</td>
<td>p ← dQNaN vxinz_flag ← 1</td>
<td>p ← dQNaN vxinz_flag ← 1</td>
<td>p ← dQNaN vxinz_flag ← 1</td>
<td>p ← dQNaN vxinz_flag ← 1</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
<tr>
<td>(-NZF)</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
<tr>
<td>(+NZF)</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>p ← M(src1,src3)</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
<tr>
<td>QNaN</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src1</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
<tr>
<td>SNaN</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← Q(src1) vxsnan_flag ← 1</td>
<td>p ← src3</td>
<td>p ← src3</td>
</tr>
</tbody>
</table>

Part 2: Subtract

<table>
<thead>
<tr>
<th>src2</th>
<th>(-\infty)</th>
<th>(-NZF)</th>
<th>(-Zero)</th>
<th>(+Zero)</th>
<th>(+NZF)</th>
<th>(+\infty)</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>v ← dQNaN vxix_flag ← 1</td>
<td>v ← dQNaN vxix_flag ← 1</td>
<td>v ← dQNaN vxix_flag ← 1</td>
<td>v ← dQNaN vxix_flag ← 1</td>
<td>v ← dQNaN vxix_flag ← 1</td>
<td>v ← dQNaN vxix_flag ← 1</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>(-NZF)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>(-Zero)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>(+Zero)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>(+NZF)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← src2</td>
<td>v ← src2</td>
</tr>
<tr>
<td>QNaN &amp; src1 is a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
<tr>
<td>QNaN &amp; src1 not a NaN</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
<td>v ← p</td>
</tr>
</tbody>
</table>

Explanation:

src1 The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ [0,1]).
src2 For xvnmsubdp, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ [0,1]).
src3 For xvnmsubadp, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ [0,1]).
dQNaN Default quiet NaN (x97F8_0000_0000_0000).
NZF Nonzero finite number.
Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number operands.
Q(x) Return a QNaN with the payload of x.
S(x,y) Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y) Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p The intermediate product having unbounded range and precision.
v The intermediate result having unbounded range and precision.
VSX Vector Negative Multiply-Subtract Single-Precision XX3-form

do i=0 to 127 by 32
reset_xflags();
src1 ← VSR[XA]{i:i+31};
src2 ← "xvnmsubasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31};
src3 ← "xvnmsubasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31};
v[0:inf] ← MultiplyAddSP(src1,src3,NegateSP(src2));
result{i:i+31} ← NegateSP(RoundToSP(RN,v));
if(vxsnan_flag) then SetFX(VXSNAN);
if(vximz_flag)  then SetFX(VXIMZ);
if(vxisi_flag)  then SetFX(VXISI);
if(ox_flag)     then SetFX(OX);
if(ux_flag)     then SetFX(UX);
if(xx_flag)     then SetFX(XX);
ex_flag ← ex_flag | (VE & vxsnan_flag);
ex_flag ← ex_flag | (VE & vximz_flag);
ex_flag ← ex_flag | (VE & vxisi_flag);
ex_flag ← ex_flag | (OE & ox_flag);
ex_flag ← ex_flag | (UE & ux_flag);
ex_flag ← ex_flag | (XE & xx_flag);
end

if( ex_flag = 0 ) then VSR[XT] ← result

For xvnmsubasp, do the following.
– Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
– Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
– Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

src1 is multiplied\(^1\) by src3, producing a product having unbounded range and precision.

See part 1 of Table 108.

src2 is negated and added\(^2\) to the product, producing a sum having unbounded range and precision.

The sum is normalized\(^3\).

See part 2 of Table 108.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 105, “Vector Floating-Point Final Result with Negation,” on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX OX UX XX VXSNAN VXISI VXIMZ

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

For xvnmsubasp, do the following.
– Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
– Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
– Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
### VSR Data Layout for `xvnmsub(al)m`sp

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = <code>xvnmsubasp</code> ? VSR[XT] : VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src3 = <code>xvnmsubasp</code> ? VSR[XB] : VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>
## Table 108. Actions for `xvnmsub(alm)`sp

<table>
<thead>
<tr>
<th>src3</th>
<th>–Infinity</th>
<th>–NZF</th>
<th>–Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
</table>
| –Infinity | p ← +Infinity | p ← +Infinity | p ← dQNaN
vximz_flag ← 1 | p ← dQNaN
vximz_flag ← 1 | p ← +Infinity | p ← –Infinity | p ← src3 | p ← Q(src3)
vxnan_flag ← 1 |
| –NZF | p ← +Infinity | p ← M(src1,src3) | p ← src1 | p ← src1 | p ← +Infinity | p ← src3 | p ← Q(src3)
vxnan_flag ← 1 |
| –Zero | p ← dQNaN vximz_flag ← 1 | p ← +Zero | p ← +Zero | p ← +Zero | p ← dQNaN
vximz_flag ← 1 | p ← src3 | p ← src3 | p ← src3 |
vximz_flag ← 1 | p ← src3 | p ← src3 | p ← src3 |
| +NZF | p ← –Infinity | p ← M src1,src3 | p ← src1 | p ← src1 | p ← +Infinity | p ← src3 | p ← Q(src3)
vxnan_flag ← 1 |
| +Infinity | p ← –Infinity | p ← +Infinity | p ← dQNaN vximz_flag ← 1 | p ← dQNaN vximz_flag ← 1 | p ← +Infinity | p ← src3 | p ← Q(src3)
vxnan_flag ← 1 |
| QNaN | p ← src1 | p ← src1 | p ← src1 | p ← src1 | p ← src1 | p ← src1 | p ← src1 | p ← src1 |
| SNaN | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 | p ← Q(src1) vxnan_flag ← 1 |

### Explanation:

- **src1**: The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
- **src2**: The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
- **src3**: The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
- **dQNaN**: Default quiet NaN (0x7FC0_0000).
- **NZF**: Nonzero finite number.
- **Rezd**: Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
- **Q(x)**: Return a QNaN with the payload of x.
- **S(x,y)**: Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
  - **Note**: If x = –y, v is considered to be an exact-zero-difference result (Rezd).
- **M(x,y)**: Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
- **p**: The intermediate product having unbounded range and precision.
- **v**: The intermediate result having unbounded range and precision.
VSX Vector Round to Double-Precision Integer using round to Nearest Away

**XX2-form**

\[ \text{xvrdpi} \] \[ XT, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>201</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
XT & \leftarrow TX || T \\
XB & \leftarrow BX || B \\
ex\_flag & \leftarrow 0b0 \\
\end{align*}
\]

\[
do \ i = 0 \ to \ 127 \ by \ 64 \\
\text{reset\_flags()} \\
\text{result} \{i:i+63\} \leftarrow \text{RoundToDPIntegerNearAway}(VSR[XB]\{i:i+63\}) \\
\text{if}(\text{vxsnan\_flag}) \ \text{then} \ \text{SetFX(VXSNAN)} \\
ex\_flag \leftarrow \text{ex\_flag} \ | \ (\text{VE} \ & \ \text{vxsnan\_flag}) \\
\text{end}
\]

\[
\text{if} (\text{ex\_flag} = 0) \ \text{then} \ VSR[XT] \leftarrow \text{result}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX VXSNAN

**VSR Data Layout for xvrdpi**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>

VSX Vector Round to Double-Precision Integer Exact using Current rounding mode

**XX2-form**

\[ \text{xvrdpic} \] \[ XT, XB \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>235</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>30</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
XT & \leftarrow TX || T \\
XB & \leftarrow BX || B \\
ex\_flag & \leftarrow 0b0 \\
\end{align*}
\]

\[
do \ i = 0 \ to \ 127 \ by \ 64 \\
\text{reset\_flags()} \\
src\{0:63\} \leftarrow \text{VSR}[XB]\{i:i+63\} \\
\text{if}(\text{RN}=0b00) \ \text{then} \\
\text{result}\{i:i+63\} \leftarrow \text{RoundToDPIntegerNearEven}(src) \\
\text{if}(\text{RN}=0b01) \ \text{then} \\
\text{result}\{i:i+63\} \leftarrow \text{RoundToDPIntegerTrunc}(src) \\
\text{if}(\text{RN}=0b10) \ \text{then} \\
\text{result}\{i:i+63\} \leftarrow \text{RoundToDPIntegerCeil}(src) \\
\text{if}(\text{RN}=0b11) \ \text{then} \\
\text{result}\{i:i+63\} \leftarrow \text{RoundToDPIntegerFloor}(src) \\
\text{if}(\text{vxsnan\_flag}) \ \text{then} \ \text{SetFX(VXSNAN)} \\
\text{if}(\text{xx\_flag}) \ \text{then} \ \text{SetFX(XX)} \\
ex\_flag \leftarrow \text{ex\_flag} \ | \ (\text{VE} \ & \ \text{vxsnan\_flag}) \\
ex\_flag \leftarrow \text{ex\_flag} \ | \ (\text{VE} \ & \ \text{xx\_flag}) \\
\text{end}
\]

\[
\text{if} (\text{ex\_flag} = 0) \ \text{then} \ VSR[XT] \leftarrow \text{result}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**

FX XX VXSNAN

**VSR Data Layout for xvrdpic**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |
VSX Vector Round to Double-Precision Integer using round toward -Infinity  XX2-form

\[
xvrdpim \quad XT, XB
\]
\[
\begin{array}{cccccc}
\hline
0 & 60 & T & /// & B & 249 \\
\hline
X & TX \mid T \mid & & & & \\
X & BX \mid B \mid & & & & \\
\hline
\end{array}
\]

\[
\text{Let } XT \text{ be the value } TX \text{ concatenated with } T. \\
\text{Let } XB \text{ be the value } BX \text{ concatenated with } B.
\]

For each vector element \( i \) from 0 to 1, do the following.

Let \( src \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[XB].

\[
src \text{ is rounded to an integer using the rounding mode Round toward } -\text{Infinity.}
\]

The result is placed into doubleword element \( i \) of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

\[
\begin{array}{c}
FX \quad VXSNAN
\end{array}
\]

VSR Data Layout for xvrdpim

\[
\begin{array}{cc}
\text{src} = \text{VSR[XB]} & \\
\text{tgt} = \text{VSR[XT]} & \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
0 & 64 & 127 \\
\hline
DP & DP & \\
\end{array}
\]

VSX Vector Round to Double-Precision Integer using round toward +Infinity  XX2-form

\[
xvrdpip \quad XT, XB
\]
\[
\begin{array}{cccccc}
\hline
0 & 60 & T & /// & B & 233 \\
\hline
X & TX \mid T \mid & & & & \\
X & BX \mid B \mid & & & & \\
\hline
\end{array}
\]

\[
\text{Let } XT \text{ be the value } TX \text{ concatenated with } T. \\
\text{Let } XB \text{ be the value } BX \text{ concatenated with } B.
\]

For each vector element \( i \) from 0 to 1, do the following.

Let \( src \) be the double-precision floating-point operand in doubleword element \( i \) of VSR[XB].

\[
src \text{ is rounded to an integer using the rounding mode Round toward } +\text{Infinity.}
\]

The result is placed into doubleword element \( i \) of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

\[
\begin{array}{c}
FX \quad VXSNAN
\end{array}
\]

VSR Data Layout for xvrdpip

\[
\begin{array}{cc}
\text{src} = \text{VSR[XB]} & \\
\text{tgt} = \text{VSR[XT]} & \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
0 & 64 & 127 \\
\hline
DP & DP & \\
\end{array}
\]
VSX Vector Round to Double-Precision Integer using round toward Zero  \textit{XX2-form}

\texttt{xvrdpiz} XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>217</th>
<th>B'X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>|</td>
<td>T</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>B</td>
<td>BX</td>
<td>|</td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\texttt{ex\_flag} \leftarrow \texttt{0b0}

\texttt{do i=0 to 127 by 64}
\begin{enumerate}
\item \texttt{reset\_xflags()}
\item \texttt{result{[i:i+63]} \leftarrow \text{RoundToSFIntegerTrunc(VSR[XB]{[i:i+63]})}}
\item \texttt{if(vxsnan\_flag) then \text{SetFX(VXSNAN)}}
\item \texttt{ex\_flag \leftarrow ex\_flag | (VE \& vxsnan\_flag)}
\end{enumerate}
\texttt{end}

\texttt{if( ex\_flag = 0 ) then VSR[XT] \leftarrow result}

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element \texttt{i} from 0 to 1, do the following.

Let \texttt{src} be the double-precision floating-point operand in doubleword element \texttt{i} of VSR[XB].

\texttt{src} is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element \texttt{i} of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

\textbf{Special Registers Altered}
\begin{itemize}
\item FX \texttt{VXSNAN}
\end{itemize}

\textbf{VSR Data Layout for xvrdpiz}
\begin{itemize}
\item \texttt{src = VSR[XB]}
\item \texttt{tgt = VSR[XT]}
\end{itemize}
VSX Vector Reciprocal Estimate
Double-Precision XX2-form

\[ x_{\text{vredp}} \rightarrow \text{XT}, \text{XB} \]

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>218</th>
<th>B'X</th>
<th>Addr</th>
</tr>
</thead>
</table>

\[ \text{XT} \leftarrow \text{TX} \| T \]
\[ \text{XB} \leftarrow \text{BX} \| B \]
\[ \text{ex\_flag} \leftarrow 0b0 \]

\[
\text{do \ i=0 \ to \ 127 \ by \ 64} \\
\text{\hspace{1cm} reset\_xflags()} \\
\text{\hspace{1cm} v(0::\text{inf}) \leftarrow \text{ReciprocalEstimateDP(VSR[XB](i:i+63))} \\
\text{\hspace{1cm} result(i:i+63) \leftarrow \text{RoundToDP}(RN,v)} \\
\text{\hspace{1cm} if(\text{vxsnan\_flag}) \text{then SetFX(VXSNAN)} } \\
\text{\hspace{1cm} if(\text{ox\_flag}) \text{then SetFX(OX)} } \\
\text{\hspace{1cm} if(\text{ux\_flag}) \text{then SetFX(UX)} } \\
\text{\hspace{1cm} if(\text{zx\_flag}) \text{then SetFX(ZX)} } \\
\text{\hspace{1cm} \text{ex\_flag} \leftarrow \text{ex\_flag} \| (\text{VE} \& \text{vxsnan\_flag}) } \\
\text{\hspace{1cm} \text{ex\_flag} \leftarrow \text{ex\_flag} \| (\text{OE} \& \text{ox\_flag}) } \\
\text{\hspace{1cm} \text{ex\_flag} \leftarrow \text{ex\_flag} \| (\text{UE} \& \text{ux\_flag}) } \\
\text{\hspace{1cm} \text{ex\_flag} \leftarrow \text{ex\_flag} \| (\text{ZE} \& \text{zx\_flag}) } \\
\text{\hspace{1cm} end} \\
\text{\hspace{1cm} if( \text{ex\_flag} = 0 ) \text{then VSR[XT] \leftarrow result} } \\
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

\[
\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-\text{Infinity}</td>
<td>-\text{Zero}</td>
<td>None</td>
</tr>
<tr>
<td>-\text{Zero}</td>
<td>-\text{Infinity}</td>
<td>ZX</td>
</tr>
<tr>
<td>+\text{Zero}</td>
<td>+\text{Infinity}</td>
<td>ZX</td>
</tr>
<tr>
<td>+\text{Infinity}</td>
<td>+\text{Zero}</td>
<td>None</td>
</tr>
<tr>
<td>\text{SNaN}</td>
<td>\text{QNaN}</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>\text{QNaN}</td>
<td>\text{QNaN}</td>
<td>None</td>
</tr>
</tbody>
</table>

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered
\[ \text{FX OX UX ZX VXSNAN} \]

VSR Data Layout for xvredp
\[ \text{src} = \text{VSR[XB]} \]
\[ \text{tgt} = \text{VSR[XT]} \]

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td></td>
<td>DP</td>
</tr>
</tbody>
</table>
VSX Vector Reciprocal Estimate
Single-Precision XX2-form

Let $XT$ be the value $TX$ concatenated with $T$.
Let $XB$ be the value $BX$ concatenated with $B$.

For each vector element $i$ from 0 to 3, do the following.
Let $src$ be the single-precision floating-point operand in word element $i$ of $VSR[XB]$.

A single-precision floating-point estimate of the reciprocal of $src$ is placed into word element $i$ of $VSR[XT]$ in single-precision format.

Unless the reciprocal of $src$ would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of $src$. That is,

$$\left| \frac{\text{estimate} - \frac{1}{src}}{\frac{1}{src}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>None</td>
</tr>
<tr>
<td>$-\text{Zero}$</td>
<td>$-\infty^1$</td>
<td>ZX</td>
</tr>
<tr>
<td>$+\text{Zero}$</td>
<td>$+\infty^1$</td>
<td>ZX</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+\text{Zero}$</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN²</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN²</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if $ZE=1$.
2. No result if $VE=1$.

If a trap-enabled exception occurs in any element of the vector, no results are written to $VSR[XT]$.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered
$FX$ $OX$ $UX$ $ZX$ $VXSNAN$

VSR Data Layout for $xvresp$

<table>
<thead>
<tr>
<th>src = $VSR[XB]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
<tr>
<td>tgt = $VSR[XT]$</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

| 0 | 32 | 64 | 96 | 127 |

Operation with various special values of the operand is summarized below.
VSX Vector Round to Single-Precision Integer
using round to Nearest Away  XX2-form

\[
xvrspi \quad XT, XB
\]

\[
\begin{array}{cccccc}
60 & T & /// & B & 137 & 60 \\
6 & 1 & 16 & 21 & \text{SP} & 60 \\
\end{array}
\]

\[
XT \leftarrow TX || T \\
XB \leftarrow BX || B \\
ex\_flag \leftarrow 0b0 \\
\]

\[
do\ i=0\ to\ 127\ by\ 32 \\
\begin{array}{l}
\text{reset\_xflags()}
\end{array}
\]

\[
\begin{array}{l}
\text{result}(i:i+31) \leftarrow \text{RoundToSPIntegerNearAway(VSR}[XB]{i:i+31})
\end{array}
\]

\[
\begin{array}{l}
\text{if(vxsnan\_flag) then SetFX(VXSNAN)}
\end{array}
\]

\[
ex\_flag \leftarrow ex\_flag \mid (VE \& vxsnan\_flag)
\]

\[
\end
\]

\[
\begin{array}{l}
\text{if( ex\_flag} = 0 \text{ ) then VSR}[XT] \leftarrow \text{result}
\end{array}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

\[
src \text{ is rounded to an integer using the rounding mode Round to Nearest Away.}
\]

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX  VXSNAN

VSR Data Layout for xvrspi
src = VSR[XB]

\[
\begin{array}{cccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP}
\end{array}
\]

tgt = VSR[XT]

\[
\begin{array}{cccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP}
\end{array}
\]

VSX Vector Round to Single-Precision Integer
Exact using Current rounding mode  XX2-form

\[
xvrspic \quad XT, XB
\]

\[
\begin{array}{cccccc}
60 & T & /// & B & 171 & 60 \\
6 & 1 & 16 & 21 & \text{SP} & 60 \\
\end{array}
\]

\[
XT \leftarrow TX || T \\
XB \leftarrow BX || B \\
ex\_flag \leftarrow 0b0 \\
\]

\[
do\ i=0\ to\ 127\ by\ 32 \\
\begin{array}{l}
\text{reset\_xflags()}
\end{array}
\]

\[
\begin{array}{l}
\text{src}(0:31) \leftarrow VSR[XB](i:i+31)
\end{array}
\]

\[
\begin{array}{l}
\text{if(RN=0b00) then RoundToSPIntegerNearEven(src)}
\end{array}
\]

\[
\begin{array}{l}
\text{if(RN=0b01) then RoundToSPIntegerTrunc(src)}
\end{array}
\]

\[
\begin{array}{l}
\text{if(RN=0b10) then RoundToSPIntegerCeil(src)}
\end{array}
\]

\[
\begin{array}{l}
\text{if(RN=0b11) then RoundToSPIntegerFloor(src)}
\end{array}
\]

\[
\begin{array}{l}
\text{if(vxsnan\_flag) then SetFX(VXSNAN)}
\end{array}
\]

\[
\begin{array}{l}
\text{if(xx\_flag) then SetFX(XX)}
\end{array}
\]

\[
ex\_flag \leftarrow ex\_flag \mid (VE \& vxsnan\_flag)
\]

\[
ex\_flag \leftarrow ex\_flag \mid (XE \& xx\_flag)
\]

\[
\end
\]

\[
\begin{array}{l}
\text{if( ex\_flag} = 0 \text{ ) then VSR}[XT] \leftarrow \text{result}
\end{array}
\]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

\[
src \text{ is rounded to an integer value using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.}
\]

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX  XX  VXSNAN

VSR Data Layout for xvrspic
src = VSR[XB]

\[
\begin{array}{cccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP}
\end{array}
\]

tgt = VSR[XT]

\[
\begin{array}{cccc}
\text{SP} & \text{SP} & \text{SP} & \text{SP}
\end{array}
\]
**VSX Vector Round to Single-Precision Integer using round toward -Infinity**  
**XX2-form**

```plaintext
xvrspim XT,XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ /</th>
<th>B</th>
<th>185</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>21</td>
<td>16</td>
<td>30</td>
</tr>
</tbody>
</table>

```plaintext
XT ← TX || T
XB ← BX || B
ex_flag ← 0b0
```

```plaintext
do i=0 to 127 by 32
    reset_xflags()
    result{i:i+31} = RoundToSPIntegerFloor(VSR[BX]{i:i+31})
    if(vxsnan_flag) then SetFX(VXSNAN)
    ex_flag ← ex_flag | (VE & vxsnan_flag)
end
```

```plaintext
if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.  
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.  
Let src be the single-precision floating-point operand in word element i of VSR[XB].

- src is rounded to an integer using the rounding mode Round toward -Infinity.
- The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**  
FX VXSNAN

**VSR Data Layout for xvrspim**

```plaintext
src = VSR[XB]
```

```
SP SP SP SP
```

```plaintext
tgt = VSR[XT]
```

```
SP SP SP SP
```

| 0  | 32 | 64 | 96 | 127 |

---

**VSX Vector Round to Single-Precision Integer using round toward +Infinity**  
**XX2-form**

```plaintext
xvrspip XT,XB
```

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ /</th>
<th>B</th>
<th>169</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>21</td>
<td>16</td>
<td>30</td>
</tr>
</tbody>
</table>

```plaintext
XT ← TX || T
XB ← BX || B
ex_flag ← 0b0
```

```plaintext
do i=0 to 127 by 32
    reset_xflags()
    result{i:i+31} = RoundToSPIntegerCeil(VSR[XB]{i:i+31})
    if(vxsnan_flag) then SetFX(VXSNAN)
    ex_flag ← ex_flag | (VE & vxsnan_flag)
end
```

```plaintext
if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.  
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.  
Let src be the single-precision floating-point operand in word element i of VSR[XB].

- src is rounded to an integer using the rounding mode Round toward +Infinity.
- The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

**Special Registers Altered**  
FX VXSNAN

**VSR Data Layout for xvrspip**

```plaintext
src = VSR[XB]
```

```
SP SP SP SP
```

```plaintext
tgt = VSR[XT]
```

```
SP SP SP SP
```

| 0  | 32 | 64 | 96 | 127 |
VSX Vector Round to Single-Precision Integer using round toward Zero  XX2-form

xvrspiz  XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>153</th>
<th>BTX</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0</td>
<td>152</td>
</tr>
</tbody>
</table>

XT ← TX || T
XB ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
reset_xflags()
result(i:i+31) ← RoundToSPIntegerTrunc(VSR[XB][i:i+31])
if(vxsnan_flag) then SetFX(VXSNAN)
ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered
FX  VXSNAN

VSR Data Layout for xvrspiz

src = VSR[XB]

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>
tgt = VSR[XT]

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
</table>
0 32 64 96 127

VSX Vector Reciprocal Square Root Estimate Double-Precision  XX2-form

xvrsqrdp  XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>202</th>
<th>BTX</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>0</td>
<td>201</td>
</tr>
</tbody>
</table>

XT ← TX || T
XB ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
reset_xflags()
v[0:inf] ← RecipSquareRootEstimateDP(VSR[XB][i:i+63])
result[i:i+63] ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(zx_flag) then SetFX(ZX)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxsqrt_flag)
ex_flag ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

\[
\frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \leq \frac{1}{16384}
\]

Operation with various special values of the operand is summarized below.
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

**Special Registers Altered**
- FX
- ZX
- VXSNAN
- VXSQRT

**VSR Data Layout for xvrsqrtedp**

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>–Finite</td>
<td>QNaN(^1)</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Zero</td>
<td>–Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity(^2)</td>
<td>ZX</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN(^1)</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.
VSX Vector Reciprocal Square Root Estimate Single-Precision $XX2$-form

xvrsqrtesp XT,XB

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>///</th>
<th>B</th>
<th>138</th>
<th>X</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
reset_xflags()
v{0:inf} ← RecipSquareRootEstimateSP(VSR[XB][i:i+31])
result[i:i+31] ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(zx_flag) then SetFX(ZX)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxsqrt_flag)
ex_flag ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{1}{\sqrt{\text{src}}} - \text{estimate} \right| \leq \frac{1}{16384}$$

Architecture Note

Increasing the required accuracy of xvrsqrtesp to 13 bits is currently under consideration.

Operation with various special values of the operand is summarized below.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>–Infinity</td>
<td>QNaN$^1$</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Zero</td>
<td>None</td>
</tr>
<tr>
<td>–Finite</td>
<td>QNaN$^1$</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>–Zero</td>
<td>–Infinity$^2$</td>
<td>ZX</td>
</tr>
<tr>
<td>+Zero</td>
<td>+Infinity$^2$</td>
<td>ZX</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN$^1$</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtesp

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP SP SP SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP SP SP SP</td>
</tr>
</tbody>
</table>

Operation with various special values of the operand is summarized below.
**VSX Vector Square Root Double-Precision XX2-form**

**xvsqrtdp**  XT,XB  

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th></th>
<th>B</th>
<th>203</th>
<th>B7X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td></td>
<td>B</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**XT** ← TX || T  
**XB** ← BX || B  
**ex_flag** ← 0b0  

**ex_flag** ← (VE & vxsnan_flag)  
**ex_flag** ← (VE & vxsqrt_flag)  
**ex_flag** ← (XE & xx_flag)  

if( ex_flag ) then VSR[XT] ← result

**Explanation:**

- **src** The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ (0,1)).
- **dQNaN** Default quiet NaN (0x7FF8_0000_0000_0000).
- **NZF** Nonzero finite number.
- **SQRT(x)** The unbounded-precision square root of the floating-point value x.
- **Q(x)** Return a QNaN with the payload of x.
- **v** The intermediate result having unbounded significand precision and unbounded exponent range.

**Table 109: Actions for xvsqrtdp**
VSX Vector Square Root Single-Precision
XX2-form

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 110.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtsp

<table>
<thead>
<tr>
<th>src = VSR[XB]</th>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
</tr>
</tbody>
</table>

| 0 | 32 | 64 | 96 | 127 |

Table 110. Actions for xvsqrtsp

<table>
<thead>
<tr>
<th>src</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>v←dQNaN</td>
<td>v←dQNaN</td>
<td>v←+Zero</td>
<td>v←+Zero</td>
<td>v←SQRT(src)</td>
<td>v←+Infinity</td>
<td>v←src</td>
<td>v←Q(src)</td>
<td>v←xvsqrt_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:

src The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN Default quiet NaN (0x7FC0_0000).
NZF Nonzero finite number.
SQRT(x) The unbounded-precision square root of the floating-point value x.
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.
Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

VSX Vector Subtract Double-Precision XXX-form

xvsubdp XT,XA,XB

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>104</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 64
reset_xflags()
src1 ← VSR[XA][i:i+63]
src2 ← VSR[XB][i:i+63]
v[0:inf] ← AddDP(src1, NegateDP(src2))
result[i:i+63] ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxisi_flag)
ex_flag ← ex_flag | (OE & ox_flag)
ex_flag ← ex_flag | (UE & ux_flag)
ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src2 is negated and added\(^1\) to src1, producing a sum having unbounded range and precision.

The sum is normalized\(^2\).

See Table 111.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

---

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>vxsnan_flag ← 1</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN vxsnan_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
src1 The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2 The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN Default quiet NaN (0x7FF8_0000_0000_0000).
NZF Nonzero finite number.
Rezd Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y) Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.

Table 111. Actions for xvsudp
VSX Vector Subtract Single-Precision
XX3-form

xvsubsp XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>72</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT</td>
<td>← TX</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex_flag</td>
<td>← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do i=0 to 127 by 32
reset_xflags()

src1 ← VSR[XA]{i:i+31}
src2 ← VSR[XB]{i:i+31}
v{0:inf} ← AddSP(src1,NegateSP(src2))
result{i:i+31} ← RoundToSP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag) then SetFX(VXISI)
if(ox_flag) then SetFX(OX)
if(ux_flag) then SetFX(UX)
if(xx_flag) then SetFX(XX)
ex_flag ← ex_flag | (VE & vxsnan_flag)
ex_flag ← ex_flag | (VE & vxisi_flag)
ex_flag ← ex_flag | (OE & ox_flag)
ex_flag ← ex_flag | (UE & ux_flag)
ex_flag ← ex_flag | (XE & xx_flag)
end

if( ex_flag ) then VSR[XT] ← result

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src2 is negated and added[1] to src1, producing a sum having unbounded range and precision.
The sum is normalized[2].

See Table 112.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.
<table>
<thead>
<tr>
<th>src2</th>
<th>-Infinity</th>
<th>-NZF</th>
<th>-Zero</th>
<th>+Zero</th>
<th>+NZF</th>
<th>+Infinity</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Infinity</td>
<td>v ← dQNaN</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>-Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← -Zero</td>
<td>v ← Rezd</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Zero</td>
<td>v ← +Infinity</td>
<td>v ← -src2</td>
<td>v ← Rezd</td>
<td>v ← +Zero</td>
<td>v ← -src2</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+NZF</td>
<td>v ← +Infinity</td>
<td>v ← S(src1,src2)</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← S(src1,src2)</td>
<td>v ← -Infinity</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>+Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← +Infinity</td>
<td>v ← dQNaN vxsi_flag ← 1</td>
<td>v ← src2</td>
<td>v ← Q(src2) vxsnan_flag ← 1</td>
</tr>
<tr>
<td>QNaN</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1</td>
<td>v ← src1 vxsnan_flag ← 1</td>
</tr>
<tr>
<td>SNaN</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
<td>v ← Q(src1) vxsnan_flag ← 1</td>
</tr>
</tbody>
</table>

Explanation:
src1 The single-precision floating-point value in word element i of VSR[XA] (where i ∈ (0,1,2,3)).
src2 The single-precision floating-point value in word element i of VSR[XB] (where i ∈ (0,1,2,3)).
dQNaN Default quiet NaN (0x7FC0_0000).
NZF Nonzero finite number.
NZF Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y) Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
Note: If x + y, v is considered to be an exact-zero-difference result (Rezd).
Q(x) Return a QNaN with the payload of x.
v The intermediate result having unbounded significand precision and unbounded exponent range.

Table 112. Actions for xvsubsp
VSX Vector Test for software Divide Double-Precision X33-form

\texttt{xvtdivdp BF,XA,XB}

<table>
<thead>
<tr>
<th>60</th>
<th>BF</th>
<th>6 (//)</th>
<th>A</th>
<th>16</th>
<th>B</th>
<th>21</th>
<th>125</th>
<th>AX</th>
<th>BX</th>
</tr>
</thead>
</table>

\[\text{XA} \gets AX || A\]
\[\text{XB} \gets BX || B\]
\[\text{eq\_flag} \gets 0b0\]
\[\text{gt\_flag} \gets 0b0\]

\text{do } i = 0 \text{ to } 127 \text{ by } 64\\
\text{src1} \gets \text{VSR}[\text{XA}][i:i+63]\\
\text{src2} \gets \text{VSR}[\text{XB}][i:i+63]\\
\text{e_a} \gets \text{src1}[1:11] - 1023\\
\text{e_b} \gets \text{src2}[1:11] - 1023\\
\text{fe\_flag} \gets \text{fe\_flag} \lor \text{IsNaN(src1)} \lor \text{IsInf(src1)} \lor \text{IsZero(src1)} \lor \text{IsNaN(src2)} \lor \text{IsInf(src2)} \lor \text{IsZero(src2)} \lor (\text{e_b} \leq -1022) \lor (\text{e_b} > 1021) \\
\text{fg\_flag} \gets \text{fg\_flag} \lor \text{IsInf(src1)} \lor \text{IsInf(src2)} \lor \text{IsZero(src2)} \lor \text{IsDen(src2)}\\
\text{fl\_flag} \gets \text{xvredp\_error()} \leq 2^{14}\\
\text{CR}[BF] \gets 0b1 || \text{fg\_flag} || \text{fe\_flag} || 0b0\\

\text{Let XA be the value AX concatenated with A.}\\
\text{Let XB be the value BX concatenated with B.}\\

\text{fe\_flag} \text{ is initialized to 0.}\\
\text{fg\_flag} \text{ is initialized to 0.}\\

\text{For each vector element } i \text{ from 0 to 127, do the following.}\\
\text{Let src1 be the double-precision floating-point operand in doubleword element } i \text{ of VSR[XA].}\\
\text{Let src2 be the double-precision floating-point operand in doubleword element } i \text{ of VSR[XB].}\\
\text{Let e_a be the unbiased exponent of src1.}\\
\text{Let e_b be the unbiased exponent of src2.}\\

\text{fe\_flag} \text{ is set to } 1 \text{ for any of the following conditions.}\\
\text{– src1 is a NaN or an infinity.}\\
\text{– src2 is a zero, a NaN, or an infinity.}\\
\text{– e_b is less than or equal to -1022.}\\
\text{– e_b is greater than or equal to 1021.}\\
\text{– src1 is not a zero and the difference, } e_a - e_b , \text{ is greater than or equal to 1023.}\\
\text{– src1 is not a zero and the difference, } e_a - e_b , \text{ is less than or equal to -1021.}\\
\text{– src1 is not a zero and } e_a \text{ is less than or equal to -970.}\\

\text{fg\_flag} \text{ is set to 1 for any of the following conditions.}\\
\text{– src1 is an infinity.}\\
\text{– src2 is a zero, an infinity, or a denormalized value.}\\

\text{CR field BF is set to the value 0b1 || fg\_flag || fe\_flag || 0b0.}\\

\textbf{Special Registers Altered}\\
\text{CR[BF]}\\

\textbf{VSR Data Layout for xvtdivdp}\\
\text{src1 = VSR[XA]}\\
\begin{array}{ll}
0 & .dword[0] \\
64 & .dword[1]
\end{array}\\
\text{src2 = VSR[XB]}\\
\begin{array}{ll}
0 & .dword[0] \\
64 & .dword[1]
\end{array}
VSX Vector Test for software Divide
Single-Precision XX3-form

\texttt{xvtdivsp BF,XA,XB}

<table>
<thead>
<tr>
<th>i</th>
<th>60</th>
<th>BF</th>
<th>A</th>
<th>B</th>
<th>93</th>
<th>0x00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XA</td>
<td>← AX</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XB</td>
<td>← BX</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eq_flag ← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gt_flag ← 0b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{do } & \text{i = 0 to 127 by 32} \\
src1 & \leftarrow \text{VSR[XA][i:i+31]} \\
src2 & \leftarrow \text{VSR[XB][i:i+31]} \\
e_a & \leftarrow \text{src1[1:8]} - 127 \\
e_b & \leftarrow \text{src2[1:8]} - 127 \\
\text{fe_flag} & \leftarrow \text{fe_flag | IsNaN(src1) | IsInf(src1) | IsNaN(src2) | IsInf(src2) | IsZero(src2) | (e_b <= -126) | (e_b >= 125) | (IsZero(src1) & (e_a - e_b == 127)) | (IsZero(src1) & (e_a - e_b <= -125)) | (IsZero(src1) & (e_a <= -103))} \\
\text{fg_flag} & \leftarrow \text{fg_flag | IsInf(src1) | IsInf(src2) | IsZero(src2) | IsDen(src2)} \\
\end{align*}

\text{end}

\text{fl_flag} ← \text{xvredp_error()} <= 2''\text{14}
\text{CR[BF]} ← 0b1 || fg_flag || fe_flag || 0b0

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

\text{fe_flag} \text{ is initialized to } 0.
\text{fg_flag} \text{ is initialized to } 0.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

Let e_a be the unbiased exponent of src1.

Let e_b be the unbiased exponent of src2.

\text{fe_flag} \text{ is set to } 1 \text{ for any of the following conditions.}
\begin{itemize}
  \item src1 is a NaN or an infinity.
  \item src2 is a zero, a NaN, or an infinity.
  \item e_b is less than or equal to -126.
  \item e_b is greater than or equal to 125.
  \item src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 127.
  \item src1 is not a zero and the difference, e_a - e_b, is less than or equal to -125.
  \item src1 is not a zero and e_a is less than or equal to -103.
\end{itemize}

\text{fg_flag} \text{ is set to } 1 \text{ for any of the following conditions.}
\begin{itemize}
  \item src1 is an infinity.
  \item src2 is a zero, an infinity, or a denormalized value.
\end{itemize}

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

\textbf{Special Registers Altered}

\textbf{CR[BF]}

\begin{tabular}{c|cccc}
\text{src1 = VSR[XA]} & \text{.word[0]} & \text{.word[1]} & \text{.word[2]} & \text{.word[3]} \\
\text{src2 = VSR[XB]} & \text{.word[0]} & \text{.word[1]} & \text{.word[2]} & \text{.word[3]} \\
\end{tabular}

0 60 93 0x00
**VSX Vector Test for software Square Root Double-Precision XX2-form**

*xvtsqrtdp  BF,XB*

<table>
<thead>
<tr>
<th>60</th>
<th>579</th>
<th>60</th>
<th>579</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>8</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>66</td>
<td>21</td>
<td>234</td>
</tr>
<tr>
<td>0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
</tbody>
</table>

 XB ← BX || B
 fe_flag ← 0b0
 fg_flag ← 0b0

do i=0 to 127 by 64
 src ← VSR[XB]{i:i+63}
 e_b ← src2{1:11} - 1023
 fe_flag ← fe_flag | IsNaN(src) | IsInf(src) |
| IsZero(src) | IsNeg(src) | ( e_a <= -970 )
 fg_flag ← fg_flag | IsInf(src) | IsZero(src) |
| IsDen(src) |
end

fl_flag ← xvrsqrtdp.error() <= 2^-14
CR[BF] ← 0b1 || fg_flag || fe_flag || 0b0

Let XB be the value BX concatenated with B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 1, do the following.
Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.
  – src is a zero, a NaN, an infinity, or a negative value.
  – e_b is less than or equal to -970.

fg_flag is set to 1 for the following condition.
  – src is a zero, an infinity, or a denormalized value.

CR field BF is set to the value
0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered
CR[BF]

**VSR Data Layout for xvtsqrtdp**

src = VSR[XB]

<table>
<thead>
<tr>
<th>60</th>
<th>579</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

| 0 | 64 | 127 |

**VSX Vector Test for software Square Root Single-Precision XX2-form**

*xvtsqrtsp BF,XB*

<table>
<thead>
<tr>
<th>60</th>
<th>579</th>
<th>60</th>
<th>579</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>8</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>66</td>
<td>21</td>
<td>170</td>
</tr>
<tr>
<td>0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
</tbody>
</table>

 XB ← BX || B
 fe_flag ← 0b0
 fg_flag ← 0b0

do i=0 to 127 by 32
 src ← VSR[XB]{i:i+31}
 e_b ← src2{1:8} - 127
 fe_flag ← fe_flag | IsNaN(src) | IsInf(src) |
| IsZero(src) | IsNeg(src) | ( e_a <= -103 )
 fg_flag ← fg_flag | IsInf(src) | IsZero(src) |
| IsDen(src) |
end

fl_flag = xvrsqrtesp.error() <= 2^-14
CR[BF] = 0b1 || fg_flag || fe_flag || 0b0

Let XB be the value BX concatenated with B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 3, do the following.
Let src be the single-precision floating-point operand in word element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.
  – src is a zero, a NaN, an infinity, or a negative value.
  – e_b is less than or equal to -103.

fg_flag is set to 1 for the following condition.
  – src is a zero, an infinity, or a denormalized value.

CR field BF is set to the value
0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered
CR[BF]

**VSR Data Layout for xvtsqrtsp**

src = VSR[XB]

<table>
<thead>
<tr>
<th>60</th>
<th>579</th>
</tr>
</thead>
</table>

| 0 | 32 | 64 | 96 | 127 |
VSX Logical AND XX3-form

xxland XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>130</th>
<th>VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $XT \leftarrow TX \| T$
- $XA \leftarrow AX \| A$
- $XB \leftarrow BX \| B$
- $VSR[XT] \leftarrow VSR[XA] \& VSR[XB]$

Let $XT$ be the value $TX$ concatenated with $T$.
Let $XA$ be the value $AX$ concatenated with $A$.
Let $XB$ be the value $BX$ concatenated with $B$.

The contents of $VSR[XA]$ are ANDed with the contents of $VSR[XB]$ and the result is placed into $VSR[XT]$.

Special Registers Altered
None

VSR Data Layout for xxland

- $src1 = VSR[XA]$
- $src2 = VSR[XB]$
- $tgt = VSR[XT]$

VSX Logical AND with Complement XX3-form

xxlandc XT,XA,XB

<table>
<thead>
<tr>
<th>0</th>
<th>60</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>138</th>
<th>VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $XT \leftarrow TX \| T$
- $XA \leftarrow AX \| A$
- $XB \leftarrow BX \| B$
- $VSR[XT] \leftarrow VSR[XA] \& \neg VSR[XB]$

Let $XT$ be the value $TX$ concatenated with $T$.
Let $XA$ be the value $AX$ concatenated with $A$.
Let $XB$ be the value $BX$ concatenated with $B$.

The contents of $VSR[XA]$ are ANDed with the complement of the contents of $VSR[XB]$ and the result is placed into $VSR[XT]$.

Special Registers Altered
None

VSR Data Layout for xxlandc

- $src1 = VSR[XA]$
- $src2 = VSR[XB]$
- $tgt = VSR[XT]$

0 127
### VSX Logical Equivalence XX3-form

**xxleqv**  
XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>186</th>
<th>0</th>
</tr>
</thead>
</table>

\[ VSR[32\times TX+T] \leftarrow VSR[32\times AX+A] = VSR[32\times BX+B] \]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

**Special Registers Altered:**
None

**VSR Data Layout for xxleqv**

<table>
<thead>
<tr>
<th>src = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src = VSR[XB]</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

### VSX Logical NAND XX3-form

**xxnand**  
XT,XA,XB

<table>
<thead>
<tr>
<th>60</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>178</th>
<th>0</th>
</tr>
</thead>
</table>

\[ VSR[32\times TX+T] \leftarrow \neg( VSR[32\times AX+A] \& VSR[32\times BX+B] ) \]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

**Special Registers Altered:**
None

**VSR Data Layout for xxnand**

<table>
<thead>
<tr>
<th>src = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>src = VSR[XB]</td>
</tr>
<tr>
<td>tgt = VSR[XT]</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
**VSX Logical OR with Complement XX3-form**

\[
\text{xxlorc} \quad \text{XT,XA,XB}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
0 & 60 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XA \) be the value \( AX \) concatenated with \( A \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

The contents of \( VSR[XA] \) are ORed with the complement of the contents of \( VSR[XB] \) and the result is placed into \( VSR[XT] \).

**Special Registers Altered:**
None

**VSR Data Layout for xxlorc**

\[
\begin{align*}
\text{src}1 &= VSR[XA] \\
\text{src}2 &= VSR[XB] \\
\text{tgt} &= VSR[XT]
\end{align*}
\]

**VSX Logical NOR XX3-form**

\[
\text{xxlnor} \quad \text{XT,XA,XB}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
0 & 60 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XA \) be the value \( AX \) concatenated with \( A \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

The contents of \( VSR[XA] \) are ORed with the contents of \( VSR[XB] \) and the complemented result is placed into \( VSR[XT] \).

**Special Registers Altered**
None

**VSR Data Layout for xxlnor**

\[
\begin{align*}
\text{src}1 &= VSR[XA] \\
\text{src}2 &= VSR[XB] \\
\text{tgt} &= VSR[XT]
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
0 & 60 & 6 & 11 & 16 & 21 \\
\hline
\end{array}
\]
VSX Logical OR XX3-form

\[ \text{xxlor} \quad XT,XA,XB \]

\[
\begin{array}{cccccc}
60 & 5 & 6 & 11 & 16 & 21 \\
0 & A & B & 146 & 154 & 160 \end{array}
\]

\[ \text{VSR}[32X+T] \leftarrow \text{VSR}[32AX+A] \lor \text{VSR}[32BX+B] \]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XA \) be the value \( AX \) concatenated with \( A \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

The contents of \( \text{VSR}[XA] \) are ORed with the contents of \( \text{VSR}[XB] \) and the result is placed into \( \text{VSR}[XT] \).

Special Registers Altered
None

VSR Data Layout for xxlor

- \( \text{src1} = \text{VSR}[XA] \)
- \( \text{src2} = \text{VSR}[XB] \)
- \( \text{tgt} = \text{VSR}[XT] \)

VSX Logical XOR XX3-form

\[ \text{xxlxor} \quad XT,XA,XB \]

\[
\begin{array}{cccccc}
60 & 5 & 6 & 11 & 16 & 21 \\
0 & A & B & 154 & 160 \end{array}
\]

\[ \text{VSR}[32X+T] \leftarrow \text{VSR}[32AX+A] \land \text{VSR}[32BX+B] \]

Let \( XT \) be the value \( TX \) concatenated with \( T \).
Let \( XA \) be the value \( AX \) concatenated with \( A \).
Let \( XB \) be the value \( BX \) concatenated with \( B \).

The contents of \( \text{VSR}[XA] \) are exclusive-ORed with the contents of \( \text{VSR}[XB] \) and the result is placed into \( \text{VSR}[XT] \).

Special Registers Altered
None

VSR Data Layout for xxlxor

- \( \text{src1} = \text{VSR}[XA] \)
- \( \text{src2} = \text{VSR}[XB] \)
- \( \text{tgt} = \text{VSR}[XT] \)
VSX Merge High Word  XX3-form

`xxmrghw`  XT,XA,XB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[0]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[0]

Let XT be the value TX concatenated with T.
LetXA be the value AX concatenated with A.
LetXB be the value BX concatenated with B.

The contents of word element 0 of VSR[XA] are placed into word element 0 of VSR[XT].
The contents of word element 0 of VSR[XB] are placed into word element 1 of VSR[XT].
The contents of word element 1 of VSR[XA] are placed into word element 2 of VSR[XT].
The contents of word element 1 of VSR[XB] are placed into word element 3 of VSR[XT].

Special Registers Altered
None

VSR Data Layout for `xxmrghw`

```
src1 = VSR[XA]
  .word[0] .word[1] unused unused

src2 = VSR[XB]
  .word[0] .word[1] unused unused
tgt = VSR[XT]
```

VSX Merge Low Word  XX3-form

`xxmrglw`  XT,XA,XB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>T</td>
<td>A</td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[2]

Let XT be the value TX concatenated with T.
LetXA be the value AX concatenated with A.
LetXB be the value BX concatenated with B.

The contents of word element 2 of VSR[XA] are placed into word element 0 of VSR[XT].
The contents of word element 2 of VSR[XB] are placed into word element 1 of VSR[XT].
The contents of word element 3 of VSR[XA] are placed into word element 2 of VSR[XT].
The contents of word element 3 of VSR[XB] are placed into word element 3 of VSR[XT].

Special Registers Altered
None

VSR Data Layout for `xxmrglw`

```
src1 = VSR[XA]

src2 = VSR[XB]
tgt = VSR[XT]
```
### VSX Permute Doubleword Immediate

**XX3-form**

**xxpermdi**

<table>
<thead>
<tr>
<th>XT,XA,XB,DM</th>
<th>0</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>0</th>
<th>DM</th>
<th>10</th>
<th>16</th>
<th>21</th>
<th>24</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>DM</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>24</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let $XT$ be the value $TX$ concatenated with $T$. Let $XA$ be the value $AX$ concatenated with $A$. Let $XB$ be the value $BX$ concatenated with $B$.

If $DM.bit[0]=0$, the contents of doubleword element 0 of $VSR[XA]$ are placed into doubleword element 0 of $VSR[XT]$. Otherwise the contents of doubleword element 1 of $VSR[AX]$ are placed into doubleword element 0 of $VSR[XT]$.

If $DM.bit[1]=0$, the contents of doubleword element 0 of $VSR[XB]$ are placed into doubleword element 1 of $VSR[XT]$. Otherwise the contents of doubleword element 1 of $VSR[AX]$ are placed into doubleword element 1 of $VSR[XT]$.

**Special Registers Altered**

None

**Extended Mnemonic**

<table>
<thead>
<tr>
<th>Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xxspltd$ $T,A,0$</td>
</tr>
<tr>
<td>$xxspltd$ $T,A,1$</td>
</tr>
<tr>
<td>$xxmrgld$ $T,A,B$</td>
</tr>
<tr>
<td>$xxwswapd$ $T,A$</td>
</tr>
</tbody>
</table>

**Table 113:**

**VSR Data Layout for xxpermdi**

<table>
<thead>
<tr>
<th>src1 = VSR[XA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
</tr>
<tr>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
</tr>
<tr>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
</tr>
<tr>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

### VSX Select XX4-form

**xxsel**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>T</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>3</th>
<th>26</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>10</td>
<td>AX</td>
<td>BX</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

do $i=0$ to 127

if ($VSR[32+CX+C].bit[i]=0$) then

$VSR[32+TX+T].bit[i] ← VSR[32+AX+A].bit[i]$

else

$VSR[32+TX+T].bit[i] ← VSR[32+BX+B].bit[i]$

end

Let $XT$ be the value $TX$ concatenated with $T$. Let $XA$ be the value $AX$ concatenated with $A$. Let $XB$ be the value $BX$ concatenated with $B$. Let $XC$ be the value $CX$ concatenated with $C$.

For each bit of $VSR[XC]$ that contains the value 0, the corresponding bit of $VSR[AX]$ is placed into the corresponding bit of $VSR[XT]$. Otherwise, the corresponding bit of $VSR[XB]$ is placed into the corresponding bit of $VSR[XT]$.

**Special Registers Altered**

None

**VSR Data Layout for xxsel**

<table>
<thead>
<tr>
<th>src1 = VSR[AX]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>src3 = VSR[XC]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>127</th>
</tr>
</thead>
</table>

## Table 113:

**VSR Data Layout for xxpermdi**

<table>
<thead>
<tr>
<th>src1 = VSR[AX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
</tr>
<tr>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src2 = VSR[XB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
</tr>
<tr>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tgt = VSR[XT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dword[0]</td>
</tr>
<tr>
<td>.dword[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>64</th>
<th>127</th>
</tr>
</thead>
</table>
VSX Shift Left Double by Word Immediate
XX3-form

xxsldwi XT,XA,XB,SHW

| 60 | T | A | B | 0 | Shw | 2 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|----|---|---|---|---|-----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |  |  |  |   |     |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

source.qword[0] ← VSR[32×AX+A]
source.qword[1] ← VSR[32×BX+B]
VSR[32×TX+T] ← source.word[SHW:SHW+3]

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let the source vector be the concatenation of the contents of VSR[XA] followed by the contents of VSR[XB]. Words SHW:SHW+3 of the source vector are placed into VSR[XT].

Special Registers Altered
None

VSR Data Layout for xxslldwi
src1 = VSR[XA]

src2 = VSR[XB]

tgt = VSR[XT]


VSX Splat Word
XX2-form

xxspltw XT,XB,UIM

<table>
<thead>
<tr>
<th>60</th>
<th>T</th>
<th>/ / /</th>
<th>UIM</th>
<th>B</th>
<th>164</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 32+TX+T].word[0] ← VSR[32×BX+B].word[UIM]
| 32+TX+T].word[1] ← VSR[32×BX+B].word[UIM]

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

The contents of word element UIM of VSR[XB] are replicated in each word element of VSR[XT].

Special Registers Altered
None

VSR Data Layout for xxspltw
src = VSR[XB]

tgt = VSR[XT]


0 32 64 96 127
Chapter 8. Signal Processing Engine (SPE)  
[Category: Signal Processing Engine]

8.1 Overview

The Signal Processing Engine (SPE) accelerates signal processing applications normally suited to DSP operation. This is accomplished using short vectors (two element) within 64-bit GPRs and using single instruction multiple data (SIMD) operations to perform the requisite computations. SPE also architects an Accumulator register to allow for back to back operations without loop unrolling.

8.2 Nomenclature and Conventions

Several conventions regarding nomenclature are used for SPE:

- The Signal Processing Engine category is abbreviated as SPE.
- Bits 0 to 31 of a 64-bit register are referenced as upper word, even word or high word element of the register. Bits 32:63 are referred to as lower word, odd word or low word element of the register. Each half is an element of a 64-bit GPR.
- Bits 0 to 15 and bits 32 to 47 are referenced as even halfwords. Bits 16 to 31 and bits 48 to 63 are referenced as odd halfwords.
- Mnemonics for SPE instructions generally begin with the letters ‘ev’ (embedded vector).

The RTL conventions in described below are used in addition to those described in Section 1.3: Additional RTL functions are described in Appendix D.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_sf</td>
<td>Signed fractional multiplication. Result of multiplying 2 signed fractional quantities having bit length n taking the least significant 2n-1 bits of the sign extended product and concatenating a 0 to the least significant bit forming a signed fractional result of 2n bits. Two 16-bit signed fractional quantities, a and b are multiplied, as shown below: ea_0:31 = EXTS(a) and eb_0:31 = EXTS(b) prod_0:63 = ea X eb eprod_0:63 = EXTS(prod_0:63) result_0:31 = eprod_33:63</td>
</tr>
</tbody>
</table>

Guarded signed fractional multiplication. Result of multiplying 2 signed fractional quantities having bit length 16 taking the least significant 31 bits of the sign extended product and concatenating a 0 to the least significant bit forming a guarded signed fractional result of 64 bits. Since guarded signed fractional multiplication produces a 64-bit result, fractional input quantities of -1 and -1 can produce +1 in the intermediate product. Two 16-bit fractional quantities, a and b are multiplied, as shown below:

ea\_0:31 = EXTS(a) and eb\_0:31 = EXTS(b) prod\_0:63 = ea X eb eprod\_0:63 = EXTS(prod\_0:63) result\_0:63 = eprod\_1:63 || 0b0

<< Logical shift left. x << y shifts value x left by y bits, leaving zeros in the vacated bits.

>> Logical shift right. x >> y shifts value x right by y bits, leaving zeros in the vacated bits.

8.3 Programming Model

8.3.1 General Operation

SPE instructions generally take elements from one source register and operate on them with the corresponding elements of a second source register (and/or the accumulator) to produce results. Results are placed in the destination register and/or the accumulator. Instructions that are vector in nature (i.e. produce results of more than one element) provide results for each element that are independent of the computation of the other elements. These instructions can also be used to perform scalar DSP operations by ignoring the results of the upper 32-bit half of the register file.
There are no record forms of SPE instructions. As a result, the meaning of bits in the CR is different than for other categories. SPE Compare instructions specify a CR field, two source registers, and the type of compare: greater than, less than, or equal. Two bits of the CR field are written with the result of the vector compare, one for each element. The remaining two bits reflect the ANDing and ORing of the vector compare results.

8.3.2 GPR Registers

The SPE requires a GPR register file with thirty-two 64-bit registers. For 32-bit implementations, instructions that normally operate on a 32-bit register file access and change only the least significant 32-bits of the GPRs leaving the most significant 32-bits unchanged. For 64-bit implementations, operation of these instructions is unchanged, i.e. those instructions continue to operate on the 64-bit registers as they would if the SPE was not implemented. Most SPE instructions view the 64-bit register as being composed of a vector of two elements, each of which is 32 bits wide (some instructions read or write 16-bit elements). The most significant 32-bits are called the upper word, high word or even word. The least significant 32-bits are called the lower word, low word or odd word. Unless otherwise specified, SPE instructions write all 64-bits of the destination register.

8.3.3 Accumulator Register

A partially visible accumulator register (ACC) is provided for some SPE instructions. The accumulator is a 64-bit register that holds the results of the Multiply Accumulate (MAC) forms of SPE Fixed-Point instructions. The accumulator allows the back-to-back execution of dependent MAC instructions, something that is found in the inner loops of DSP code such as FIR and FFT filters. The accumulator is partially visible to the programmer in the sense that its results do not have to be explicitly read to use them. Instead they are always copied into a 64-bit destination GPR which is specified as part of the instruction. Based upon the type of instruction, the accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.
FXH represents the logical ‘or’ of all the bits shifted right from the Guard bit when the fractional result is normalized. FXH is zeroed if an overflow, underflow, or invalid input error is detected on the high element of an SPE.Embedded Float Vector instruction.

Execution of an SPE.Embedded Float Scalar instruction leaves FXH undefined.

36 Embedded Floating-Point Invalid Operation/Input Error High (FINVH) [Category: SP.FV]
The FINVH bit is set to 1 if any high word operand of an SPE.Embedded Float Vector instruction is infinity, NaN, or a denormalized value, or if the instruction is a divide and the dividend and divisor are both 0, or if a conversion to integer or fractional value overflows.

Execution of an SPE.Embedded Float Scalar instruction leaves FINVH undefined.

37 Embedded Floating-Point Divide By Zero High (FDBZH) [Category: SP.FV]
The FDBZH bit is set to 1 when an SPE.Embedded Vector Floating-Point Divide instruction is executed with a divisor of 0 in the high word operand, and the dividend is a finite nonzero number.

Execution of an SPE.Embedded Float Scalar instruction leaves FDBZH undefined.

38 Embedded Floating-Point Underflow High (FUNFH) [Category: SP.FV]
The FUNFH bit is set to 1 when the execution of an SPE.Embedded Float Vector instruction results in an underflow on the high word operation.

Execution of an SPE.Embedded Float Scalar instruction leaves FUNFH undefined.

39 Embedded Floating-Point Overflow High (FOVFH) [Category: SP.FV]
The FOVFH bit is set to 1 when the execution of an SPE.Embedded Float Vector instruction results in an overflow on the high word operation.

Execution of an SPE.Embedded Float Scalar instruction leaves FOVFH undefined.

40:41 Reserved

42 Embedded Floating-Point Inexact Sticky Flag (FINXS) [Categories: SP.FV, SP.FD, SP.FS]
The FINXS bit is set to 1 whenever the execution of an Embedded Floating-Point instruction delivers an inexact result for either the low or high element and no Embedded Floating-Point Data interrupt is taken for either element, or if an Embedded Floating-Point instruction results in overflow (FOVF=1 or FOVFH=1), but Embedded Floating-Point Overflow exceptions are disabled (FOVFE=0), or if an Embedded Floating-Point instruction results in underflow (FUNF=1 or FUNFH=1), but Embedded Floating-Point Underflow exceptions are disabled (FUNFE=0), and no Embedded Floating-Point Data interrupt occurs. This is a sticky bit.

43 Embedded Floating-Point Invalid Operation/Input Sticky Flag (FINVS) [Categories: SP.FV, SP.FD, SP.FS]
The FINVS bit is defined to be the sticky result of any Embedded Floating-Point instruction that causes FINVH or FINV to be set to 1. That is, FINVS ← FINVS | FINV | FINVH. This is a sticky bit.

44 Embedded Floating-Point Divide By Zero Sticky Flag (FDBZS) [Categories: SP.FV, SP.FD, SP.FS]
The FDBZS bit is set to 1 when an Embedded Floating-Point Divide instruction sets FDBZH or FDBZ to 1. That is, FDBZS ← FDBZS | FDBZ | FDBZH. This is a sticky bit.

45 Embedded Floating-Point Underflow Sticky Flag (FUNFS) [Categories: SP.FV, SP.FD, SP.FS]
The FUNFS bit is defined to be the sticky result of any Embedded Floating-Point instruction that causes FUNFH or FUNF to be set to 1. That is, FUNFS ← FUNFS | FUNF | FUNFH. This is a sticky bit.

46 Embedded Floating-Point Overflow Sticky Flag (FOVFS) [Categories: SP.FV, SP.FD, SP.FS]
The FOVFS bit is defined to be the sticky result of any Embedded Floating-Point instruction that causes FOVH or FOVF to be set to 1. That is, FOVFS ← FOVFS | FOVF | FOVFH. This is a sticky bit.

47 Summary Integer Overflow (SOV)
SOV is set to 1 when an SPE instruction sets OV to 1. This is a sticky bit.

48 Integer Overflow (OV)
OV is set to 1 to indicate that an overflow has occurred in the lower element during execution of an SPE instruction. The bit is set to 1 if a result of an operation performed by the instruction cannot be represented in the number of bits into which the result is to be placed, and is set to 0 otherwise. The OV bit is not altered by Modulo instructions, or by other instructions that cannot overflow.

49 Embedded Floating-Point Guard Bit (Low/scalar) (FG) [Categories: SP.FV, SP.FD, SP.FS]
FG is supplied for use by the Embedded Floating-Point Round interrupt handler. FG is an extension of the low-order bits of the fractional result produced from an Embedded Floating-Point instruction on the low word. FG is zeroed if an overflow, underflow, or invalid input error is detected on the low element of an Embedded Floating-Point instruction.

**51 Embedded Floating-Point Inexact Bit (Low/scalar)** (FX) [Categories: SP.FV, SPFD, SPFS]
FX is supplied for use by the Embedded Floating-Point Round interrupt handler. FX is an extension of the low-order bits of the fractional result produced from an Embedded Floating-Point instruction on the low word. FX represents the logical 'or' of all the bits shifted right from the Guard bit when the fractional result is normalized. FX is zeroed if an overflow, underflow, or invalid input error is detected on Embedded Floating-Point instruction.

**52 Embedded Floating-Point Invalid Operation/Input Error (Low/scalar)** (FINV) [Categories: SP.FV, SPFD, SPFS]
The FINV bit is set to 1 if any low word operand of an Embedded Floating-Point instruction is infinity, NaN, or a denormalized value, or if the operation is a divide and the dividend and divisor are both 0, or if a conversion to integer or fractional value overflows.

**53 Embedded Floating-Point Divide By Zero (Low/scalar)** (FDBZ) [Categories: SP.FV, SPFD, SPFS]
The FDBZ bit is set to 1 when an Embedded Floating-Point Divide instruction is executed with a divisor of 0 in the low word operand, and the dividend is a finite nonzero number.

**54 Embedded Floating-Point Underflow (Low/scalar)** (FUNF) [Categories: SP.FV, SPFD, SPFS]
The FUNF bit is set to 1 when the execution of an Embedded Floating-Point instruction results in an underflow on the low word operation.

**55 Embedded Floating-Point Overflow (Low/scalar)** (FOVF) [Categories: SP.FV, SPFD, SPFS]
The FOVF bit is set to 1 when the execution of an Embedded Floating-Point instruction results in an overflow on the low word operation.

**56 Reserved**

**57 Embedded Floating-Point Round (Inexact) Exception Enable** (FINXE) [Categories: SP.FV, SPFD, SPFS]

0 Exception disabled
1 Exception enabled

The Embedded Floating-Point Round interrupt is taken if the exception is enabled and if FG | FGH | FX | FXH (signifying an inexact result) is set to 1 as a result of an Embedded Floating-Point instruction.

If an Embedded Floating-Point instruction results in overflow or underflow and the corresponding Embedded Floating-Point Underflow or Embedded Floating-Point Overflow exception is disabled then the Embedded Floating-Point Round interrupt is taken.

**58 Embedded Floating-Point Invalid Operation/Input Error Exception Enable** (FINVE) [Categories: SP.FV, SPFD, SPFS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FINV or FINVH bit is set to 1 by an Embedded Floating-Point instruction.

**59 Embedded Floating-Point Divide By Zero Exception Enable** (FDBZE) [Categories: SP.FV, SPFD, SPFS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FDBZ or FDBZH bit is set to 1 by an Embedded Floating-Point instruction.

**60 Embedded Floating-Point Underflow Exception Enable** (FUNFE) [Categories: SP.FV, SPFD, SPFS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FUNF or FUNFH bit is set to 1 by an Embedded Floating-Point instruction.

**61 Embedded Floating-Point Overflow Exception Enable** (FOVFE) [Categories: SP.FV, SPFD, SPFS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FOVF or FOVFH bit is set to 1 by an Embedded Floating-Point instruction.

**62:63 Embedded Floating-Point Rounding Mode Control** (FRMC) [Categories: SP.FV, SPFD, SPFS]

00 Round to Nearest
Programming Note

Rounding modes 0b10 (+Infinity) and 0b11 (-Infinity) may not be supported by some implementations. If an implementation does not support these, Embedded Floating-Point Round interrupts are generated for every Embedded Floating-Point instruction for which rounding is required when +Infinity or -Infinity modes are set and software is required to produce the correctly rounded result.

8.3.5 Data Formats

The SPE provides two different data formats, integer and fractional. Both data formats can be treated as signed or unsigned quantities.

8.3.5.1 Integer Format

Unsigned integers consist of 16, 32, or 64-bit binary integer values. The largest representable value is $2^{n-1}$ where \( n \) represents the number of bits in the value. The smallest representable value is 0. Computations that produce values larger than $2^{n-1}$ or smaller than 0 may set OV or OVH in the SPEFSCR.

Signed integers consist of 16, 32, or 64-bit binary values in two's-complement form. The largest representable value is $2^{n-1}-1$ where \( n \) represents the number of bits in the value. The smallest representable value is $-2^{n-1}$. Computations that produce values larger than $2^{n-1}-1$ or smaller than $-2^{n-1}$ may set OV or OVH in the SPEFSCR.

8.3.5.2 Fractional Format

Fractional data format is conventionally used for DSP fractional arithmetic. Fractional data is useful for representing data converted from analog devices.

Unsigned fractions consist of 16, 32, or 64-bit binary fractional values that range from 0 to less than 1. Unsigned fractions place the radix point immediately to the left of the most significant bit. The most significant bit of the value represents the value $2^{-1}$, the next most significant bit represents the value $2^{-2}$ and so on. The largest representable value is $1-2^{-32}$ where \( n \) represents the number of bits in the value. The smallest representable value is 0. Computations that produce values larger than $1-2^{-n}$ or smaller than 0 may set OV or OVH in the SPEFSCR. The SPE category does not define unsigned fractional forms of instructions to manipulate unsigned fractional data since the unsigned integer forms of the instructions produce the same results as would the unsigned fractional forms.

Guarded unsigned fractions are 64-bit binary fractional values. Guarded unsigned fractions place the decimal point immediately to the left of bit 32. The largest representable value is $2^{32}-2^{-32}$. The smallest representable value is 0. Guarded unsigned fractional computations are always modulo and do not set OV or OVH in the SPEFSCR.

Signed fractions consist of 16, 32, or 64-bit binary fractional values in two's-complement form that range from -1 to less than 1. Signed fractions place the decimal point immediately to the right of the most significant bit. The largest representable value is $1-2^{-32+n-1}$ where \( n \) represents the number of bits in the value. The smallest representable value is -1. Computations that produce values larger than $1-2^{-32+n-1}$ or smaller than -1 may set OV or OVH in the SPEFSCR. Multiplication of two signed fractional values causes the result to be shifted left one bit to remove the resultant redundant sign bit in the product. In this case, a 0 bit is concatenated as the least significant bit of the shifted result.

Guarded signed fractions are 64-bit binary fractional values. Guarded signed fractions place the decimal point immediately to the left of bit 33. The largest representable value is $2^{32}-2^{-32}$. The smallest representable value is $-2^{-32+1}$. Guarded signed fractional computations are always modulo and do not set OV or OVH in the SPEFSCR.

8.3.6 Computational Operations

The SPE category supports several different computational capabilities. Both modulo and saturation results can be performed. Modulo results produce truncation of the overflow bits in a calculation, therefore overflow does not occur and no saturation is performed. For instructions for which overflow occurs, saturation provides a maximum or minimum representable value (for the data type) in the case of overflow. Instructions are provided for a wide range of computational capability. The operation types can be divided into 4 basic categories:

- **Simple Vector** instructions. These instructions use the corresponding low and high word elements of the operands to produce a vector result that is placed in the destination register, the accumulator, or both.

- **Multiply and Accumulate** instructions. These instructions perform multiply operations, optionally add the result to the accumulator, and place the result into the destination register and optionally into the accumulator. These instructions are composed of different multiply forms, data formats and data accumulate options. The mnemonics for these instructions indicate their various characteristics. These are shown in Table 114.

- **Load** and **Store** instructions. These instructions provide load and store capabilities for moving data...
to and from memory. A variety of forms are provided that position data for efficient computation.

- Compare and miscellaneous instructions. These instructions perform miscellaneous functions such as field manipulation, bit reversed incrementing, and vector compares.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Multiply Form</strong></td>
<td></td>
</tr>
<tr>
<td>he</td>
<td>halfword even</td>
<td>16 X 16 → 32</td>
</tr>
<tr>
<td>heg</td>
<td>halfword even guarded</td>
<td>16 X 16 → 32, 64-bit final accumulate result</td>
</tr>
<tr>
<td>ho</td>
<td>halfword odd</td>
<td>16 X 16 → 32</td>
</tr>
<tr>
<td>hog</td>
<td>halfword odd guarded</td>
<td>16 X 16 → 32, 64-bit final accumulate result</td>
</tr>
<tr>
<td>w</td>
<td>word</td>
<td>32 X 32 → 64</td>
</tr>
<tr>
<td>wh</td>
<td>word high</td>
<td>32 X 32 → 32 (high-order 32 bits of product)</td>
</tr>
<tr>
<td>wl</td>
<td>word low</td>
<td>32 X 32 → 32 (low-order 32 bits of product)</td>
</tr>
<tr>
<td></td>
<td><strong>Data Format</strong></td>
<td></td>
</tr>
<tr>
<td>smf</td>
<td>signed modulo fractional</td>
<td>modulo, no saturation or overflow</td>
</tr>
<tr>
<td>smi</td>
<td>signed modulo integer</td>
<td>modulo, no saturation or overflow</td>
</tr>
<tr>
<td>ssf</td>
<td>signed saturate fractional</td>
<td>saturation on product and accumulate</td>
</tr>
<tr>
<td>ssi</td>
<td>signed saturate integer</td>
<td>saturation on product and accumulate</td>
</tr>
<tr>
<td>umi</td>
<td>unsigned modulo integer</td>
<td>modulo, no saturation or overflow</td>
</tr>
<tr>
<td>usi</td>
<td>unsigned saturate integer</td>
<td>saturation on product and accumulate</td>
</tr>
<tr>
<td></td>
<td><strong>Accumulate Option</strong></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>place in accumulator</td>
<td>result → accumulator</td>
</tr>
<tr>
<td>aa</td>
<td>add to accumulator</td>
<td>accumulator + result → accumulator</td>
</tr>
<tr>
<td>aaw</td>
<td>add to accumulator as word elements</td>
<td>accumulator&lt;sub&gt;0:31&lt;/sub&gt; + result&lt;sub&gt;0:31&lt;/sub&gt; → accumulator&lt;sub&gt;0:31&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>accumulator&lt;sub&gt;32:63&lt;/sub&gt; + result&lt;sub&gt;32:63&lt;/sub&gt; → accumulator&lt;sub&gt;32:63&lt;/sub&gt;</td>
</tr>
<tr>
<td>an</td>
<td>add negated to accumulator</td>
<td>accumulator - result → accumulator</td>
</tr>
<tr>
<td>anw</td>
<td>add negated to accumulator as word elements</td>
<td>accumulator&lt;sub&gt;0:31&lt;/sub&gt; - result&lt;sub&gt;0:31&lt;/sub&gt; → accumulator&lt;sub&gt;0:31&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>accumulator&lt;sub&gt;32:63&lt;/sub&gt; - result&lt;sub&gt;32:63&lt;/sub&gt; → accumulator&lt;sub&gt;32:63&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
8.3.7 SPE Instructions

8.3.8 Saturation, Shift, and Bit
Reverse Models

For saturation, left shifts, and bit reversal, the pseudo
RTL is provided here to more accurately describe those
functions that are referenced in the instruction pseudo
RTL.

8.3.8.1 Saturation

SATURATE(ov, carry, sat_ovn, sat_ov, val)
if ov then
    if carry then
        return sat_ovn
    else
        return sat_ov
else
    return val

8.3.8.2 Shift Left

SL(value, cnt)
if cnt > 31 then
    return 0
else
    return (value << cnt)

8.3.8.3 Bit Reverse

BITREVERSE(value)
result ← 0
mask ← 1
shift ← 31
cnt ← 32
while cnt > 0 then do
t ← value & mask
if shift >= 0 then
    result ← (t << shift) | result
else
    result ← (t >> -shift) | result
    cnt ← cnt - 1
    shift ← shift - 2
    mask ← mask << 1
return result
### 8.3.9 SPE Instruction Set

#### Bit Reversed Increment \( EVX\)-form

<table>
<thead>
<tr>
<th>brinc</th>
<th>RT,RA,RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4)</td>
<td>(5)</td>
</tr>
</tbody>
</table>

\(n \leftarrow \text{implementation-dependent number of mask bits}\)

\(\text{mask} \leftarrow (\text{RB})_{64-0:63}\)

\(a \leftarrow ([\text{RA}]_{64-0:63})\)

\(d \leftarrow \text{BITREVERSE}(1 + \text{BITREVERSE}(a \mid (\neg \text{mask})))\)

\(\text{RT} \leftarrow (\text{RA})_{0:63-n} \mid (d \& \text{mask})\)

**brinc** computes a bit-reverse index based on the contents of RA and a mask specified in RB. The new index is written to RT.

The number of bits in the mask is implementation-dependent but may not exceed 32.

**Special Registers Altered:**

None

---

#### Programming Note

**brinc** provides a way for software to access FFT data in a bit-reversed manner. RA contains the index into a buffer that contains data on which FFT is to be performed. RB contains a mask that allows the index to be updated with bit-reversed addressing. Typically this instruction precedes a load with index instruction; for example,

```
brinc r2, r3, r4
lhax r8, r5, r2
```

RB contains a bit-mask that is based on the number of points in an FFT. To access a buffer containing \(n\) byte sized data that is to be accessed with bit-reversed addressing, the mask has \(\log_2 n\) 1s in the least significant bit positions and 0s in the remaining most significant bit positions. If, however, the data size is a multiple of a halfword or a word, the mask is constructed so that the 1s are shifted left by \(\log_2 (\text{size of the data})\) and 0s are placed in the least significant bit positions.

---

#### Vector Absolute Value \( EVX\)-form

<table>
<thead>
<tr>
<th>evabs</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4)</td>
<td>(5)</td>
</tr>
</tbody>
</table>

\(RT_{0:31} \leftarrow \text{ABS}((\text{RA})_{0:31})\)

\(RT_{32:63} \leftarrow \text{ABS}((\text{RA})_{32:63})\)

The absolute value of each element of RA is placed in the corresponding elements of RT. An absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000.

**Special Registers Altered:**

None

---

#### Vector Add Immediate Word \( EVX\)-form

<table>
<thead>
<tr>
<th>evaddiw</th>
<th>RT,RB,UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4)</td>
<td>(5)</td>
</tr>
</tbody>
</table>

\(RT_{0:31} \leftarrow (\text{RB})_{0:31} \oplus \text{EXTZ}(\text{UI})\)

\(RT_{32:63} \leftarrow (\text{RB})_{32:63} \oplus \text{EXTZ}(\text{UI})\)

UI is zero-extended and added to both the high and low elements of RB and the results are placed in RT. Note that the same value is added to both elements of the register.

**Special Registers Altered:**

None

---

#### Vector Add Signed, Modulo, Integer to Accumulator Word \( EVX\)-form

<table>
<thead>
<tr>
<th>evaddsmiaaw</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4)</td>
<td>(5)</td>
</tr>
</tbody>
</table>

\(RT_{0:31} \leftarrow (\text{ACC})_{0:31} + (\text{RA})_{0:31}\)

\(RT_{32:63} \leftarrow (\text{ACC})_{32:63} + (\text{RA})_{32:63}\)

\(\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}\)

Each word element in RA is added to the corresponding element in the accumulator and the results are placed in RT and into the accumulator.

**Special Registers Altered:**

\(\text{ACC}\)
Vector Add Signed, Saturate, Integer to Accumulator Word EVX-form

\[\text{evaddssiaaw RT,RA}\]

\[
\begin{array}{cccccccc}
4 & RT & RA & /// & 1217 & \text{acc} & \text{ov} & \text{ovh} \\\n\end{array}
\]

\[
temp_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{0:31}) + \text{EXTS}((\text{RA})_{0:31}) \]
\[
\text{ovh} \leftarrow \text{temp}_{0:31} \oplus \text{temp}_{32} \\
\text{RT}_{0:31} \leftarrow \text{SATURATE} \left(\text{ovh}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})
\]
\[
temp_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{32:63}) + \text{EXTS}((\text{RA})_{32:63}) \]
\[
\text{ovl} \leftarrow \text{temp}_{31} \oplus \text{temp}_{32} \\
\text{RT}_{32:63} \leftarrow \text{SATURATE} \left(\text{ovl}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})
\]

\[\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \]
\[\text{SPEFSCR_{OVH}} \leftarrow \text{ovh} \]
\[\text{SPEFSCR_{OV}} \leftarrow \text{ovl} \]
\[\text{SPEFSCR_{SOVH}} \leftarrow \text{SPEFSCR_{SOVH}} | \text{ovh} \]
\[\text{SPEFSCR_{SOV}} \leftarrow \text{SPEFSCR_{SOV}} | \text{ovl} \]

Each signed-integer word element in RA is sign-extended and added to the corresponding sign-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
- ACC
- OV
- OVH
- SOV
- SOVH

Vector Add Unsigned, Saturate, Integer to Accumulator Word EVX-form

\[\text{evaddusiaaw RT,RA}\]

\[
\begin{array}{cccccccc}
4 & RT & RA & /// & 1216 & \text{acc} & \text{ov} & \text{ovh} \\\n\end{array}
\]

\[
temp_{0:63} \leftarrow \text{EXTZ}((\text{ACC})_{0:31}) + \text{EXTZ}((\text{RA})_{0:31}) \]
\[
\text{ovh} \leftarrow \text{temp}_{0:31} \oplus \text{temp}_{31} \\
\text{RT}_{0:31} \leftarrow \text{SATURATE} \left(\text{ovh}, \text{temp}_{31}, 0xFFFF_FFFF, 0xFFFF_FFFF, \text{temp}_{32:63})
\]
\[
temp_{0:63} \leftarrow \text{EXTZ}((\text{ACC})_{32:63}) + \text{EXTZ}((\text{RA})_{32:63}) \]
\[
\text{ovl} \leftarrow \text{temp}_{31} \oplus \text{temp}_{32} \\
\text{RT}_{32:63} \leftarrow \text{SATURATE} \left(\text{ovl}, \text{temp}_{31}, 0xFFFF_FFFF, 0xFFFF_FFFF, \text{temp}_{32:63})
\]

\[\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \]
\[\text{SPEFSCR_{OVH}} \leftarrow \text{ovh} \]
\[\text{SPEFSCR_{OV}} \leftarrow \text{ovl} \]
\[\text{SPEFSCR_{SOVH}} \leftarrow \text{SPEFSCR_{SOVH}} | \text{ovh} \]
\[\text{SPEFSCR_{SOV}} \leftarrow \text{SPEFSCR_{SOV}} | \text{ovl} \]

Each unsigned-integer word element in RA is zero-extended and added to the corresponding zero-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
- ACC
- OV
- OVH
- SOV
- SOVH

Vector Add Unsigned, Modulo, Integer to Accumulator Word EVX-form

\[\text{evaddumiaaw RT,RA}\]

\[
\begin{array}{cccccccc}
4 & RT & RA & /// & 1224 & \text{acc} & \text{ov} \\\n\end{array}
\]

\[
\text{RT}_{0:31} \leftarrow (\text{ACC})_{0:31} + (\text{RA})_{0:31} \\
\text{RT}_{32:63} \leftarrow (\text{ACC})_{32:63} + (\text{RA})_{32:63} \\
\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}
\]

Each unsigned-integer word element in RA is added to the corresponding element in the accumulator and the results are placed in RT and the accumulator.

Special Registers Altered:
- ACC

Vector Add Word EVX-form

\[\text{evadw RT,RA,RB}\]

\[
\begin{array}{cccccccc}
4 & RT & RA & RB & /// & 512 & \text{acc} & \text{ov} \\\n\end{array}
\]

\[
\text{RT}_{0:31} \leftarrow (\text{RA})_{0:31} + (\text{RB})_{0:31} \\
\text{RT}_{32:63} \leftarrow (\text{RA})_{32:63} + (\text{RB})_{32:63}
\]

The corresponding elements of RA and RB are added and the results are placed in RT. The sum is a modulo sum.

Special Registers Altered:
- None
Vector AND

```
evand RT,RA,RB
```

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>529</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td>i1</td>
<td>h6</td>
<td>i1</td>
<td></td>
</tr>
</tbody>
</table>

```

\[
\begin{align*}
RT_{0:31} & \leftarrow [RA]_{0:31} \land [RB]_{0:31} \\
RT_{32:63} & \leftarrow [RA]_{32:63} \land [RB]_{32:63}
\end{align*}
\]

The corresponding elements of RA and RB are ANDed bitwise and the results are placed in the corresponding element of RT.

Special Registers Altered:
None

Vector AND with Complement

```
evandc RT,RA,RB
```

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>530</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td>i1</td>
<td>h6</td>
<td>i1</td>
<td></td>
</tr>
</tbody>
</table>

```

```
\[
\begin{align*}
RT_{0:31} & \leftarrow (RA)_{0:31} \land \neg (RB)_{0:31} \\
RT_{32:63} & \leftarrow (RA)_{32:63} \land \neg (RB)_{32:63}
\end{align*}
\]

The word elements of RA are ANDed bitwise with the complement of the corresponding elements of RB. The results are placed in the corresponding element of RT.

Special Registers Altered:
None

Vector Compare Equal

```
evcmp eq BF,RA,RB
```

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF</th>
<th>//</th>
<th>RA</th>
<th>RB</th>
<th>564</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td>b6</td>
<td>h6</td>
<td>i1</td>
<td>i1</td>
<td></td>
</tr>
</tbody>
</table>

```

```
\[
\begin{align*}
ah & \leftarrow [RA]_{0:31} \\
al & \leftarrow [RA]_{32:63} \\
bh & \leftarrow [RB]_{0:31} \\
bl & \leftarrow [RB]_{32:63} \\
\text{if}\ (ah = bh) & \text{then} \ ch \leftarrow 1 \\
\text{else} \ ch \leftarrow 0 \\
\text{if}\ (al = bl) & \text{then} \ cl \leftarrow 1 \\
\text{else} \ cl \leftarrow 0 \\
\end{align*}
\]

\[
\text{CR}4_{4\times BF+32;4\times BF+35} \leftarrow \text{ch} || \text{cl} || \text{(ch} \land \text{cl)} || \text{(ch} \lor \text{cl)}
\]

The most significant bit in BF is set if the high-order element of RA is equal to the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is equal to the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:
CR field BF

Vector Compare Greater Than Signed

```
evcmp gts BF,RA,RB
```

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF</th>
<th>//</th>
<th>RA</th>
<th>RB</th>
<th>561</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td>b6</td>
<td>h6</td>
<td>i1</td>
<td>i1</td>
<td></td>
</tr>
</tbody>
</table>

```

```
\[
\begin{align*}
ah & \leftarrow (RA)_{0:31} \\
al & \leftarrow (RA)_{32:63} \\
bh & \leftarrow (RB)_{0:31} \\
bl & \leftarrow (RB)_{32:63} \\
\text{if}\ (ah > bh) & \text{then} \ ch \leftarrow 1 \\
\text{else} \ ch \leftarrow 0 \\
\text{if}\ (al > bl) & \text{then} \ cl \leftarrow 1 \\
\text{else} \ cl \leftarrow 0 \\
\end{align*}
\]

\[
\text{CR}4_{4\times BF+32;4\times BF+35} \leftarrow \text{ch} || \text{cl} || \text{(ch} \land \text{cl)} || \text{(ch} \lor \text{cl)}
\]

The most significant bit in BF is set if the high-order element of RA is greater than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is greater than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:
CR field BF
Vector Compare Greater Than Unsigned

**EVX-form**

`evcmpgtu BF,RA,RB`

```
<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>560</th>
</tr>
</thead>
</table>
```

ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah >u bh) then ch ← 1
else ch ← 0
if (al >u bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in BF is set if the high-order element of RA is greater than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is greater than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

**Special Registers Altered:**
CR field BF

---

Vector Compare Less Than Signed

**EVX-form**

`evcmplts BF,RA,RB`

```
<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>563</th>
</tr>
</thead>
</table>
```

ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in BF is set if the high-order element of RA is less than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is less than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

**Special Registers Altered:**
CR field BF

---

Vector Compare Less Than Unsigned

**EVX-form**

`evcmpltu BF,RA,RB`

```
<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>562</th>
</tr>
</thead>
</table>
```

ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah <u bh) then ch ← 1
else ch ← 0
if (al <u bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)

The most significant bit in BF is set if the high-order element of RA is less than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is less than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

**Special Registers Altered:**
CR field BF
### Vector Count Leading Signed Bits Word

**EVX-form**

<table>
<thead>
<tr>
<th>evcntlsw</th>
<th>RT,RA</th>
</tr>
</thead>
</table>

\[
\begin{array}{cccccc}
\text{n} & \text{s} & \text{RT} & \text{RA} & \text{///} & \text{526} \\
4 & 5 & 1 & 1 & 21 & 31
\end{array}
\]

\[
n \leftarrow 0 \\
s \leftarrow |RA|_n \\
do \text{while } n < 32 \\
\quad \text{if } (RA)_n \neq s \text{ then leave} \\
\quad n \leftarrow n + 1 \\
RT_{0:31} \leftarrow n \\
n \leftarrow 0 \\
s \leftarrow |RA|_{n+32} \\
do \text{while } n < 32 \\
\quad \text{if } (RA)_{n+32} \neq s \text{ then leave} \\
\quad n \leftarrow n + 1 \\
RT_{32:63} \leftarrow n
\]

The leading sign bits in each element of RA are counted, and the respective count is placed into each element of RT.

**Special Registers Altered:** None

---

### Programming Note

`evcntlzw` is used for unsigned operands; `evcntlsw` is used for signed operands.

### Vector Count Leading Zeros Word

**EVX-form**

<table>
<thead>
<tr>
<th>evcntlzw</th>
<th>RT,RA</th>
</tr>
</thead>
</table>

\[
\begin{array}{cccccc}
\text{n} & \text{s} & \text{RT} & \text{RA} & \text{///} & \text{525} \\
4 & 5 & 1 & 1 & 21 & 31
\end{array}
\]

\[
n \leftarrow 0 \\
do \text{while } n < 32 \\
\quad \text{if } (RA)_n = 1 \text{ then leave} \\
\quad n \leftarrow n + 1 \\
RT_{0:31} \leftarrow n \\
n \leftarrow 0 \\
do \text{while } n < 32 \\
\quad \text{if } (RA)_{n+32} = 1 \text{ then leave} \\
\quad n \leftarrow n + 1 \\
RT_{32:63} \leftarrow n
\]

The leading zero bits in each element of RA are counted, and the respective count is placed into each element of RT.

**Special Registers Altered:** None

---

### Vector Divide Word Signed

**EVX-form**

<table>
<thead>
<tr>
<th>evdivws</th>
<th>RT,RA,RB</th>
</tr>
</thead>
</table>

\[
\begin{array}{cccccc}
\text{ddh} & \text{dvl} & \text{dvh} & \text{dv} & \text{///} & \text{1222} \\
4 & 5 & 1 & 1 & 21 & 31
\end{array}
\]

\[
ddh \leftarrow |RA|_{0:31} \\
ddl \leftarrow |RA|_{32:63} \\
dvh \leftarrow |RB|_{0:31} \\
dvl \leftarrow |RB|_{32:63} \\
RT_{0:31} \leftarrow ddh + dvh \\
RT_{32:63} \leftarrow dvl + dv1 \\
ovh \leftarrow 0 \\
ovl \leftarrow 0 \\
\quad \text{if } ((ddh < 0) \& (dvh = 0)) \text{ then} \\
\quad \quad RT_{0:31} \leftarrow 0x8000_0000 \\
\quad \quad ovh \leftarrow 1 \\
\quad \text{else if } ((ddh >= 0) \& (dvh = 0)) \text{ then} \\
\quad \quad RT_{0:31} \leftarrow 0x7FFFFFFF \\
\quad \quad ovh \leftarrow 1 \\
\quad \text{else if } (ddh = 0x8000_0000)\&(dvh = 0xFFFF_FFFF) \text{ then} \\
\quad \quad RT_{0:31} \leftarrow 0x7FFFFFFF \\
\quad \quad ovh \leftarrow 1 \\
\quad \text{else if } (ddh = 0x8000_0000)\&(dv = 0xFFFF_FFFF) \text{ then} \\
\quad \quad RT_{32:63} \leftarrow 0x8000_0000 \\
\quad \quad ovl \leftarrow 1 \\
\quad \text{else if } ((dvl < 0) \& (dv1 = 0)) \text{ then} \\
\quad \quad RT_{32:63} \leftarrow 0x7FFFFFFF \\
\quad \quad ovl \leftarrow 1 \\
\quad \text{else if } ((dvl >= 0) \& (dv1 = 0)) \text{ then} \\
\quad \quad RT_{32:63} \leftarrow 0x7FFFFFFF \\
\quad \quad ovl \leftarrow 1 \\
\quad \text{else if } (dvl = 0x8000_0000)\&(dv1 = 0xFFFF_FFFF) \text{ then} \\
\quad \quad RT_{32:63} \leftarrow 0x7FFFFFFF \\
\quad \quad ovl \leftarrow 1 \\
\quad \text{SPEFSCR} \leftarrow ovh \\
\quad \text{SPEFSCR} \leftarrow ovl \\
\quad \text{SPEFSCR} \leftarrow ovh \& ovh \\
\quad \text{SPEFSCR} \leftarrow ovl \& ovl \\
\]

The two dividends are the two elements of the contents of RA. The two divisors are the two elements of the contents of RB. The resulting two 32-bit quotients on each element are placed into RT. The remainders are not supplied. The operands and quotients are interpreted as signed integers.

**Special Registers Altered:** OV OVH SOV SOVH

---

### Programming Note

Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.
Vector Divide Word Unsigned

Vector Divide Word Unsigned

\[
evdivwu \quad RT, RA, RB
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
4 & RT & RA & RB & 1223 & 31
\end{array}
\]

ddh ← (RA)0:31
ddl ← (RA)32:63
dvh ← (RB)0:31
dvl ← (RB)32:63
RT0:31 ← ddh + dvh
RT32:63 ← ddl + dvl
ovh ← 0
ovl ← 0
if (dvh = 0) then
    RT0:31 ← 0xFFFFFFFF
    ovh ← 1
if (dvl = 0) then
    RT32:63 ← 0xFFFFFFFF
    ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl

The two dividends are the two elements of the contents of RA. The two divisors are the two elements of the contents of RB. Two 32-bit quotients are formed as a result of the division on each of the high and low elements and the quotients are placed into RT. Rema

Special Registers Altered:
OV OVH SOV SOVH

Programming Note
Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Vector Equivalent

Vector Equivalent

\[
eveqv \quad RT, RA, RB
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
4 & RT & RA & RB & 537 & 31
\end{array}
\]

RT0:31 ← (RA)0:31 = (RB)0:31
RT32:63 ← (RA)32:63 = (RB)32:63

The corresponding elements of RA and RB are XORed bitwise, and the complemented results are placed in RT.

Special Registers Altered:
None

Vector Extend Sign Byte

Vector Extend Sign Byte

\[
evextsb \quad RT, RA
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
4 & RT & RA & /// & 522 & 31
\end{array}
\]

RT0:31 ← EXTS((RA)24:31)
RT32:63 ← EXTS((RA)48:63)

The signs of the low-order byte in each of the elements in RA are extended, and the results are placed in RT.

Special Registers Altered:
None

Vector Extend Sign Halfword

Vector Extend Sign Halfword

\[
evextsh \quad RT, RA
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
4 & RT & RA & /// & 523 & 31
\end{array}
\]

RT0:31 ← EXTS((RA)16:31)
RT32:63 ← EXTS((RA)48:63)

The signs of the odd halfwords in each of the elements in RA are extended, and the results are placed in RT.

Special Registers Altered:
None
**Vector Load Double Word into Double Word EVX-form**

```
evldd RT, D(RA)
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
<td>RA</td>
<td>UI</td>
<td>769</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI × 8)
RT ← MEM(EA, 8)

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

**Special Registers Altered:**

None

**Vector Load Double Word into Double Word Indexed EVX-form**

```
evlddx RT, RA, RB
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>768</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)

The doubleword addressed by EA is loaded from memory and placed in RT.

**Special Registers Altered:**

None

**Vector Load Double into Four Halfwords EVX-form**

```
evldh RT, D(RA)
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
<td>RA</td>
<td>UI</td>
<td>773</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI × 8)
RT0:15 ← MEM(EA, 2)
RT16:31 ← MEM(EA+2, 2)
RT32:47 ← MEM(EA+4, 2)
RT48:63 ← MEM(EA+6, 2)

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

**Special Registers Altered:**

None

**Vector Load Double into Four Halfwords Indexed EVX-form**

```
evldhx RT, RA, RB
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>772</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA, 2)
RT16:31 ← MEM(EA+2, 2)
RT32:47 ← MEM(EA+4, 2)
RT48:63 ← MEM(EA+6, 2)

The doubleword addressed by EA is loaded from memory and placed in RT.

**Special Registers Altered:**

None
**Vector Load Double into Two Words**  
**EVX-form**

```
evldw RT, D(RA)
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>771</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0  
else b ← (RA)  
EA ← b + EXTZ(UI×8)  
RT_{0:31} ← MEM(EA, 4)  
RT_{32:63} ← MEM(EA+4, 4)

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

**Special Registers Altered:**  
None

**Vector Load Double into Two Words**  
**Indexed**

```
evldwx RT, RA, RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>770</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
RT_{0:31} ← MEM(EA, 4)  
RT_{32:63} ← MEM(EA+4, 4)

The doubleword addressed by EA is loaded from memory and placed in RT.

**Special Registers Altered:**  
None

**Vector Load Halfword into Halfwords**  
**EVX-form**

```
evlhesplat RT, D(RA)
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>777</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0  
else b ← (RA)  
EA ← b + EXTZ(UI×2)  
RT_{0:15} ← MEM(EA, 2)  
RT_{16:31} ← 0x0000  
RT_{32:47} ← MEM(EA, 2)  
RT_{48:63} ← 0x0000

D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

**Special Registers Altered:**  
None

**Vector Load Halfword into Halfwords**  
**Indexed**

```
evlhesplatx RT, RA, RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>776</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0  
else b ← (RA)  
EA ← b + (RB)  
RT_{0:15} ← MEM(EA, 2)  
RT_{16:31} ← 0x0000  
RT_{32:47} ← MEM(EA, 2)  
RT_{48:63} ← 0x0000

The halfword addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

**Special Registers Altered:**  
None
Vector Load Halfword into Halfword Odd Signed and Splat EVX-form

evllhossplat RT,D(RA)

\[
\begin{array}{cccccc}
\text{4} & \text{RT} & \text{RA} & \text{UI} & 783 & \text{31} \\
\end{array}
\]

if \( (RA = 0) \) then \( b \leftarrow 0 \)
else \( b \leftarrow [RA] \)
\( \text{EA} \leftarrow b + \text{EXTZ}(\text{UI} \times 2) \)
\( \text{RT}_{0:31} \leftarrow \text{EXTS} (\text{MEM}(\text{EA}, 2)) \)
\( \text{RT}_{32:63} \leftarrow \text{EXTS} (\text{MEM}(\text{EA}, 2)) \)

D in the instruction mnemonic is \( \text{UI} \times 2 \). The halfword addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:
None

Vector Load Halfword into Halfword Odd Signed and Splat Indexed EVX-form

evllhossplatx RT,RA,RB

\[
\begin{array}{cccccc}
\text{4} & \text{RT} & \text{RA} & \text{RB} & 782 & \text{31} \\
\end{array}
\]

if \( (RA = 0) \) then \( b \leftarrow 0 \)
else \( b \leftarrow [RA] \)
\( \text{EA} \leftarrow b + (\text{RB}) \)
\( \text{RT}_{0:31} \leftarrow \text{EXTS} (\text{MEM}(\text{EA}, 2)) \)
\( \text{RT}_{32:63} \leftarrow \text{EXTS} (\text{MEM}(\text{EA}, 2)) \)

The halfword addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:
None

Vector Load Halfword into Halfword Odd Unsigned and Splat EVX-form

evllhousplat RT,D(RA)

\[
\begin{array}{cccccc}
\text{4} & \text{RT} & \text{RA} & \text{UI} & 781 & \text{31} \\
\end{array}
\]

if \( (RA = 0) \) then \( b \leftarrow 0 \)
else \( b \leftarrow [RA] \)
\( \text{EA} \leftarrow b + \text{EXTZ}(\text{UI} \times 2) \)
\( \text{RT}_{0:31} \leftarrow \text{EXTZ} (\text{MEM}(\text{EA}, 2)) \)
\( \text{RT}_{32:63} \leftarrow \text{EXTZ} (\text{MEM}(\text{EA}, 2)) \)

D in the instruction mnemonic is \( \text{UI} \times 2 \). The halfword addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:
None

Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed EVX-form

evllhousplatx RT,RA,RB

\[
\begin{array}{cccccc}
\text{4} & \text{RT} & \text{RA} & \text{RB} & 780 & \text{31} \\
\end{array}
\]

if \( (RA = 0) \) then \( b \leftarrow 0 \)
else \( b \leftarrow [RA] \)
\( \text{EA} \leftarrow b + (\text{RB}) \)
\( \text{RT}_{0:31} \leftarrow \text{EXTZ} (\text{MEM}(\text{EA}, 2)) \)
\( \text{RT}_{32:63} \leftarrow \text{EXTZ} (\text{MEM}(\text{EA}, 2)) \)

The halfword addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:
None
Vector Load Word into Two Halfwords

**Even**

**EVX-form**

evlwhe RT, D(RA)

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>785</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)

EA ← b + EXTZ(U×4)
RT0:15 ← MEM(EA, 2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA+2, 2)
RT48:63 ← 0x0000

D in the instruction mnemonic is U×4. The word addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

**Special Registers Altered:**

None

---

Vector Load Word into Two Halfwords

**Even Indexed**

**EVX-form**
evlwheel RT, RA, RB

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>784</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)

EA ← b + (RB)
RT0:15 ← MEM(EA, 2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA+2, 2)
RT48:63 ← 0x0000

The word addressed by EA is loaded from memory and placed in the even halfwords in each element of RT. The odd halfwords of each element of RT are set to 0.

**Special Registers Altered:**

None

---

Vector Load Word into Two Halfwords

**Odd Signed (with sign extension)**

**EVX-form**
evlwhos RT, D(RA)

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>791</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)

EA ← b + EXTZ(U×4)
RT0:31 ← EXTZ(MEM(EA, 2))
RT32:63 ← EXTZ(MEM(EA+2, 2))

D in the instruction mnemonic is U×4. The word addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

**Special Registers Altered:**

None

---

Vector Load Word into Two Halfwords

**Odd Signed Indexed (with sign extension)**

**EVX-form**
evlwhosx RT, RA, RB

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>790</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)

EA ← b + (RB)
RT0:31 ← EXTZ(MEM(EA, 2))
RT32:63 ← EXTZ(MEM(EA+2, 2))

The word addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

**Special Registers Altered:**

None
Vector Load Word into Two Halfwords
Odd Unsigned (zero-extended) **EVX-form**

```
evwhou    RT,D(RA)
```

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>789</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0
else b ← [RA]
EA ← b + EXTZ(UI × 4)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA+2,2))

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords and Splat **EVX-form**

```
evwhsplat  RT,D(RA)
```

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>797</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0
else b ← [RA]
EA ← b + EXTZ(UI × 4)
RT0:15 ← MEM(EA,2)
RT16:31 ← MEM(EA+1,2)
RT32:47 ← MEM(EA+2,2)
RT48:63 ← MEM(EA+3,2)

D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in both the even and odd halfwords in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended) **EVX-form**

```
evwhoux    RT,RA,RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>788</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0
else b ← [RA]
EA ← b + (RB)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA+2,2))

The word addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:
None

Vector Load Word into Two Halfwords and Splat Indexed **EVX-form**

```
evwhsplatx RT,RA,RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>796</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0
else b ← [RA]
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← MEM(EA+1,2)
RT32:47 ← MEM(EA+2,2)
RT48:63 ← MEM(EA+3,2)

The word addressed by EA is loaded from memory and placed in both the even and odd halfwords in each element of RT.

Special Registers Altered:
None
Vector Load Word into Word and Splat

**EVX-form**

\[
evlwwsplat \ RT, D(RA)
\]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>793</th>
</tr>
</thead>
</table>

if \( RA = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + \text{EXTZ}(UI \times 4) \)
\( RT_{0:31} \leftarrow \text{MEM}(EA, 4) \)
\( RT_{32:63} \leftarrow \text{MEM}(EA, 4) \)

The word addressed by \( EA \) is loaded from memory and placed in both elements of \( RT \).

Special Registers Altered:
None

---

**Indexed**

\[
evlwwsplatx \ RT, RA, RB
\]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>792</th>
</tr>
</thead>
</table>

if \( RA = 0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT_{0:31} \leftarrow \text{MEM}(EA, 4) \)
\( RT_{32:63} \leftarrow \text{MEM}(EA, 4) \)

The word addressed by \( EA \) is loaded from memory and placed in both elements of \( RT \).

Special Registers Altered:
None

---

Vector Merge High

**EVX-form**

\[
evmergehi \ RT, RA, RB
\]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>556</th>
</tr>
</thead>
</table>

\( RT_{0:31} \leftarrow (RA)_{0:31} \)
\( RT_{32:63} \leftarrow (RB)_{0:31} \)

The high-order elements of \( RA \) and \( RB \) are merged and placed in \( RT \).

Special Registers Altered:
None

---

Vector Merge Low

**EVX-form**

\[
evmergelo \ RT, RA, RB
\]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>557</th>
</tr>
</thead>
</table>

\( RT_{0:31} \leftarrow (RA)_{32:63} \)
\( RT_{32:63} \leftarrow (RB)_{32:63} \)

The low-order elements of \( RA \) and \( RB \) are merged and placed in \( RT \).

Special Registers Altered:
None

---

**Programming Note**

A vector splat high can be performed by specifying the same register in \( RA \) and \( RB \).

---

A vector splat low can be performed by specifying the same register in \( RA \) and \( RB \).
Vector Merge High/Low EVX-form

evmergehilo RT,RA,RB

| 4 | RT | RA | RB | 558 |

RT0:31 ← (RA)0:31
RT32:63 ← (RB)32:63

The high-order element of RA and the low-order element of RB are merged and placed in RT.

Special Registers Altered:
None

Programming Note
With appropriate specification of RA and RB, evmergehi, evmergelohi, evmergehilo, and evmergelohi provide a full 32-bit permute of two source operands.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate EVX-form

evmsmfaa RT,RA,RB

| 4 | RT | RA | RB | 1323 |

temp0:63 ← (RA)32:47 × gsf (RB)32:47
RT0:63 ← (ACC)0:63 + temp0:63
ACC0:63 ← (RT)0:63

The corresponding low even-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Note
If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Merge Low/High EVX-form

evmergelohi RT,RA,RB

| 4 | RT | RA | RB | 559 |

RT0:31 ← (RA)32:63
RT32:63 ← (RB)0:31

The low-order element of RA and the high-order element of RB are merged and placed in RT.

Special Registers Altered:
None

Programming Note
A vector swap can be performed by specifying the same register in RA and RB.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative EVX-form

evmsmfan RT,RA,RB

| 4 | RT | RA | RB | 1451 |

temp0:63 ← (RA)32:47 × gsf (RB)32:47
RT0:63 ← (ACC)0:63 - temp0:63
ACC0:63 ← (RT)0:63

The corresponding low even-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Note
If the two input operands are both -1.0, the intermediate product is represented as +1.0.
Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate EVX-form

evmhegsmiaa RT, RA, RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1321</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
temp_{0:31} \leftarrow (RA)_{0:27} \times_{ui} (RB)_{32:47}
\]
\[
temp_{0:63} \leftarrow \text{EXTZ}(temp_{0:31})
\]
\[
RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low even-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate EVX-form

evmhegumiaa RT, RA, RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1320</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
temp_{0:31} \leftarrow (RA)_{0:27} \times_{ui} (RB)_{32:47}
\]
\[
temp_{0:63} \leftarrow \text{EXTZ}(temp_{0:31})
\]
\[
RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative EVX-form

evmhegsman RT, RA, RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1449</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
temp_{0:31} \leftarrow (RA)_{0:27} \times_{ui} (RB)_{32:47}
\]
\[
temp_{0:63} \leftarrow \text{EXTS}(temp_{0:31})
\]
\[
RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low even-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended and subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative EVX-form

evmheguman RT, RA, RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1448</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
temp_{0:31} \leftarrow (RA)_{0:27} \times_{ui} (RB)_{32:47}
\]
\[
temp_{0:63} \leftarrow \text{EXTZ}(temp_{0:31})
\]
\[
RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low even-numbered unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed in RT and into the accumulator.

Special Registers Altered:
ACC
Vector Multiply Halfwords, Even, Signed, Modulo, Fractional

**EVX-form**

evmhesmf RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1035</th>
</tr>
</thead>
</table>

\[
\begin{align*}
R_{T0:31} & \leftarrow (R_{A})_{0:15} \times_{sf} (R_{B})_{0:15} \\
R_{T32:63} & \leftarrow (R_{A})_{32:47} \times_{sf} (R_{B})_{32:47}
\end{align*}
\]

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied then placed into the corresponding words of RT.

**Special Registers Altered:**

None

---

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator

**EVX-form**

evmhesmfa RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1067</th>
</tr>
</thead>
</table>

\[
\begin{align*}
R_{T0:31} & \leftarrow (R_{A})_{0:15} \times_{sf} (R_{B})_{0:15} \\
R_{T32:63} & \leftarrow (R_{A})_{32:47} \times_{sf} (R_{B})_{32:47} \\
A_{CC_{0:63}} & \leftarrow (R_{T})_{0:63}
\end{align*}
\]

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied then placed into the corresponding words of RT and into the accumulator.

**Special Registers Altered:**

ACC

---

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words

**EVX-form**

evmhesmfaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1291</th>
</tr>
</thead>
</table>

\[
\begin{align*}
t_{0:31} & \leftarrow (R_{A})_{0:15} \times_{sf} (R_{B})_{0:15} \\
R_{T0:31} & \leftarrow (A_{CC})_{0:31} + t_{0:31} \\
t_{0:31} & \leftarrow (R_{A})_{32:47} \times_{sf} (R_{B})_{32:47} \\
R_{T32:63} & \leftarrow (A_{CC})_{32:63} + t_{0:31} \\
A_{CC_{0:63}} & \leftarrow (R_{T})_{0:63}
\end{align*}
\]

For each word element in the accumulator, the corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding RT words and into the accumulator.

**Special Registers Altered:**

ACC

---

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words

**EVX-form**

evmhesfaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1419</th>
</tr>
</thead>
</table>

\[
\begin{align*}
t_{0:31} & \leftarrow (R_{A})_{0:15} \times_{sf} (R_{B})_{0:15} \\
R_{T0:31} & \leftarrow (A_{CC})_{0:31} - t_{0:31} \\
t_{0:31} & \leftarrow (R_{A})_{32:47} \times_{sf} (R_{B})_{32:47} \\
R_{T32:63} & \leftarrow (A_{CC})_{32:63} - t_{0:31} \\
A_{CC_{0:63}} & \leftarrow (R_{T})_{0:63}
\end{align*}
\]

For each word element in the accumulator, the corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32-bit intermediate products are subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding RT words and into the accumulator.

**Special Registers Altered:**

ACC
Vector Multiply Halfwords, Even, Signed, Modulo, Integer
EVX-form
evmhesmi RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1033</th>
</tr>
</thead>
</table>

\[ RT_{0:31} \leftarrow (RA)_{0:15} \times_{\text{si}} (RB)_{0:15} \]
\[ RT_{32:63} \leftarrow (RA)_{32:47} \times_{\text{si}} (RB)_{32:47} \]

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX-form
evmhesmia RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1065</th>
</tr>
</thead>
</table>

\[ RT_{0:31} \leftarrow (RA)_{0:15} \times_{\text{si}} (RB)_{0:15} \]
\[ RT_{32:63} \leftarrow (RA)_{32:47} \times_{\text{si}} (RB)_{32:47} \]
\[ ACC_{0:63} \leftarrow (RT)_{0:63} \]

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX-form
evmhesmiaaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1289</th>
</tr>
</thead>
</table>

\[ temp_{0:31} \leftarrow (RA)_{0:15} \times_{\text{si}} (RB)_{0:15} \]
\[ RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31} \]
\[ temp_{0:31} \leftarrow (RA)_{32:47} \times_{\text{si}} (RB)_{32:47} \]
\[ RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31} \]
\[ ACC_{0:63} \leftarrow (RT)_{0:63} \]

For each word element in the accumulator, the corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX-form
evmhesmianw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1417</th>
</tr>
</thead>
</table>

\[ temp_{0:31} \leftarrow (RA)_{0:15} \times_{\text{si}} (RB)_{0:15} \]
\[ RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31} \]
\[ temp_{0:31} \leftarrow (RA)_{32:47} \times_{\text{si}} (RB)_{32:47} \]
\[ RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31} \]
\[ ACC_{0:63} \leftarrow (RT)_{0:63} \]

For each word element in the accumulator, the corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
ACC
Vector Multiply Halfwords, Even, Signed, Saturate, Fractional

\[ \text{evmhessf} \quad \text{RT,RA,RB} \]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
4 & 5 & 6 & 7 & 8 & 27 \\
\hline
\end{array}
\]

\[
\text{temp}_{0:31} \leftarrow (\text{RA})_{0:15} \times_{\text{sf}} (\text{RB})_{0:15} \\
\text{if } ((\text{RA})_{0:15} = 0x8000) \& (\text{RB})_{0:15} = 0x8000 \text{ then} \\
\quad \text{RT}_{0:31} \leftarrow 0x7FFFF_FFFF \\
\quad \text{movh} \leftarrow 1 \\
\text{else} \\
\quad \text{RT}_{0:31} \leftarrow \text{temp}_{0:31} \\
\quad \text{movh} \leftarrow 0 \\
\text{temp}_{0:31} \leftarrow (\text{RA})_{32:47} \times_{\text{sf}} (\text{RB})_{32:47} \\
\text{if } ((\text{RA})_{32:47} = 0x8000) \& (\text{RB})_{32:47} = 0x8000 \text{ then} \\
\quad \text{RT}_{32:63} \leftarrow 0x7FFFF_FFFF \\
\quad \text{movl} \leftarrow 1 \\
\text{else} \\
\quad \text{RT}_{32:63} \leftarrow \text{temp}_{0:31} \\
\quad \text{movl} \leftarrow 0 \\
\text{SPEFSCR}_{\text{OVH}} \leftarrow \text{movh} \\
\text{SPEFSCR}_{\text{OV}} \leftarrow \text{movl} \\
\text{SPEFSCR}_{\text{SOVH}} \leftarrow \text{SPEFSCR}_{\text{SOVH}} \| \text{movh} \\
\text{SPEFSCR}_{\text{SOV}} \leftarrow \text{SPEFSCR}_{\text{SOV}} \| \text{movl} \\
\end{array}
\]

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
\[ \text{OV OVH SOV SOVH} \]

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator

\[ \text{evmhessfa} \quad \text{RT,RA,RB} \]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
4 & 5 & 6 & 7 & 8 & 27 \\
\hline
\end{array}
\]

\[
\text{temp}_{0:31} \leftarrow (\text{RA})_{0:15} \times_{\text{sf}} (\text{RB})_{0:15} \\
\text{if } ((\text{RA})_{0:15} = 0x8000) \& (\text{RB})_{0:15} = 0x8000 \text{ then} \\
\quad \text{RT}_{0:31} \leftarrow 0x7FFFF_FFFF \\
\quad \text{movh} \leftarrow 1 \\
\text{else} \\
\quad \text{RT}_{0:31} \leftarrow \text{temp}_{0:31} \\
\quad \text{movh} \leftarrow 0 \\
\text{temp}_{0:31} \leftarrow (\text{RA})_{32:47} \times_{\text{sf}} (\text{RB})_{32:47} \\
\text{if } ((\text{RA})_{32:47} = 0x8000) \& (\text{RB})_{32:47} = 0x8000 \text{ then} \\
\quad \text{RT}_{32:63} \leftarrow 0x7FFFF_FFFF \\
\quad \text{movl} \leftarrow 1 \\
\text{else} \\
\quad \text{RT}_{32:63} \leftarrow \text{temp}_{0:31} \\
\quad \text{movl} \leftarrow 0 \\
\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \\
\text{SPEFSCR}_{\text{OVH}} \leftarrow \text{movh} \\
\text{SPEFSCR}_{\text{OV}} \leftarrow \text{movl} \\
\text{SPEFSCR}_{\text{SOVH}} \leftarrow \text{SPEFSCR}_{\text{SOVH}} \| \text{movh} \\
\text{SPEFSCR}_{\text{SOV}} \leftarrow \text{SPEFSCR}_{\text{SOV}} \| \text{movl} \\
\end{array}
\]

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
\[ \text{ACC OV OVH SOV SOVH} \]
Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words

evmhessfaaw RT, RA, RB

4 4 RT RA RB 1283

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words

evmhessfanw RT, RA, RB

4 4 RT RA RB 1411

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH
 Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words

**EVX-form**

evmhessiaaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1281</th>
</tr>
</thead>
</table>

\[
t_{0:31} \leftarrow (RAS_{0:15} \times RBS_{0:15})
\]

\[
t_{0:63} \leftarrow \text{EXTS}(\text{ACC} _{0:31}) + \text{EXTS}(t_{0:31})
\]

\[
o_{31} \leftarrow \text{SATURATE}(o_{31}, t_{31}, 0x8000_0000, 0x7FFF_FFFF, t_{32:63})
\]

\[
t_{0:31} \leftarrow (RAS_{32:47} \times RBS_{32:47})
\]

\[
t_{0:63} \leftarrow \text{EXTS}(\text{ACC} _{32:63}) - \text{EXTS}(t_{0:31})
\]

\[
o_{31} \leftarrow \text{SATURATE}(o_{31}, t_{31}, 0x8000_0000, 0x7FFF_FFFF, t_{32:63})
\]

\[
\text{ACC} _{0:63} \leftarrow (RT) _{0:63}
\]

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

**Special Registers Altered:**

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words

**EVX-form**

evmhessianw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1409</th>
</tr>
</thead>
</table>

\[
t_{0:31} \leftarrow (RAS_{0:15} \times RBS_{0:15})
\]

\[
t_{0:63} \leftarrow \text{EXTS}(\text{ACC} _{0:31}) - \text{EXTS}(t_{0:31})
\]

\[
o_{31} \leftarrow \text{SATURATE}(o_{31}, t_{31}, 0x8000_0000, 0x7FFF_FFFF, t_{32:63})
\]

\[
t_{0:31} \leftarrow (RAS_{32:47} \times RBS_{32:47})
\]

\[
t_{0:63} \leftarrow \text{EXTS}(\text{ACC} _{32:63}) - \text{EXTS}(t_{0:31})
\]

\[
o_{31} \leftarrow \text{SATURATE}(o_{31}, t_{31}, 0x8000_0000, 0x7FFF_FFFF, t_{32:63})
\]

\[
\text{ACC} _{0:63} \leftarrow (RT) _{0:63}
\]

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

**Special Registers Altered:**

ACC OV OVH SOV SOVH
Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer EVX-form

\texttt{evmheumi} RT, RA, RB

\[
\begin{array}{c|c|c|c|c|c|c}
\text{p} & \text{RT} & \text{RA} & \text{RB} & \text{1032} \\
\hline
4 & 0 & 16 & 21 & 31 \\
\end{array}
\]

\(RT_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}\)
\(RT_{32:63} \leftarrow (RA)_{32:47} \times (RB)_{32:47}\)

The corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator EVX-form

\texttt{evmheumia} RT, RA, RB

\[
\begin{array}{c|c|c|c|c|c|c}
\text{p} & \text{RT} & \text{RA} & \text{RB} & \text{1064} \\
\hline
4 & 0 & 16 & 21 & 31 \\
\end{array}
\]

\(RT_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}\)
\(RT_{32:63} \leftarrow (RA)_{32:47} \times (RB)_{32:47}\)
\(ACC_{0:63} \leftarrow (RT)_{0:63}\)

The corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words EVX-form

\texttt{evmheumiaaw} RT, RA, RB

\[
\begin{array}{c|c|c|c|c|c|c}
\text{p} & \text{RT} & \text{RA} & \text{RB} & \text{1288} \\
\hline
4 & 0 & 16 & 21 & 31 \\
\end{array}
\]

\(temp_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}\)
\(RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}\)
\(temp_{0:31} \leftarrow (RA)_{32:47} \times (RB)_{32:47}\)
\(RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}\)
\(ACC_{0:63} \leftarrow (RT)_{0:63}\)

For each word element in the accumulator, the corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words and the sums are placed into the corresponding RT and accumulator words.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words EVX-form

\texttt{evmheumianw} RT, RA, RB

\[
\begin{array}{c|c|c|c|c|c|c}
\text{p} & \text{RT} & \text{RA} & \text{RB} & \text{1416} \\
\hline
4 & 0 & 16 & 21 & 31 \\
\end{array}
\]

\(temp_{0:31} \leftarrow (RA)_{0:15} \times (RB)_{0:15}\)
\(RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}\)
\(temp_{0:31} \leftarrow (RA)_{32:47} \times (RB)_{32:47}\)
\(RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}\)
\(ACC_{0:63} \leftarrow (RT)_{0:63}\)

For each word element in the accumulator, the corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding RT and accumulator words.

Special Registers Altered:
ACC
Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words

**EVX-form**

```
evmheusiaaw RT,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each word element in the accumulator, corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

**Special Registers Altered:**

ACC OV OVH SOV SOVH

---

Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words

**EVX-form**

```
evmheusianw RT,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1408</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each word element in the accumulator, corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

**Special Registers Altered:**

ACC OV OVH SOV SOVH
**Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate**

**EVX-form**

```
evmhogsmaa RT,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1327</th>
</tr>
</thead>
</table>

\[
temp_{0:63} \leftarrow (RA)_{48:63} \times_{\text{gsf}} (RB)_{48:63}
\]

\[
RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}
\]

\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low odd-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

**Special Registers Altered:**

ACC

---

**Note**

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

---

**Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate**

**EVX-form**

```
evmhogsmaiaa RT,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1325</th>
</tr>
</thead>
</table>

\[
temp_{0:31} \leftarrow (RA)_{48:63} \times_{\text{si}} (RB)_{48:63}
\]

\[
temp_{0:63} \leftarrow \text{EXTS}(temp_{0:31})
\]

\[
RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}
\]

\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low odd-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended to 64 bits then added to the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

**Special Registers Altered:**

ACC

---

**Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative**

**EVX-form**

```
evmhogsman RT,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1455</th>
</tr>
</thead>
</table>

\[
temp_{0:63} \leftarrow (RA)_{48:63} \times_{\text{gsf}} (RB)_{48:63}
\]

\[
RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}
\]

\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low odd-numbered halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

**Special Registers Altered:**

ACC

---

**Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative**

**EVX-form**

```
evmhogsman RT,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1453</th>
</tr>
</thead>
</table>

\[
temp_{0:31} \leftarrow (RA)_{48:63} \times_{\text{si}} (RB)_{48:63}
\]

\[
temp_{0:63} \leftarrow \text{EXTS}(temp_{0:31})
\]

\[
RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}
\]

\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low odd-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended to 64 bits then subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

**Special Registers Altered:**

ACC

---
Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate EVX-form

evmhogumiaa RT, RA, RB

```
| p | 4 | RT | RA | RB | 1324 |
```

temp0:31 ← (RA)48:63 ×ui (RB)48:63
temp0:63 ← EXTZ(temp0:31)
RT0:63 ← (ACC)0:63 + temp0:63
ACC0:63 ← (RT)0:63

The corresponding low odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended to 64 bits then added to the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
- ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional EVX-form

```evmhosmf RT, RA, RB
```

```
| p | 4 | RT | RA | RB | 1039 |
```

RT0:31 ← (RA)16:31 ×sf (RB)16:31
RT32:63 ← (RA)48:63 ×sf (RB)48:63

The corresponding odd-numbered, halfword signed fractional elements in RA and RB are multiplied. Each product is placed into the corresponding words of RT.

Special Registers Altered:
- None

Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative EVX-form

```
evmhogumian RT, RA, RB
```

```
| p | 4 | RT | RA | RB | 1452 |
```

temp0:31 ← (RA)48:63 ×ui (RB)48:63
temp0:63 ← EXTZ(temp0:31)
RT0:63 ← (ACC)0:63 - temp0:63
ACC0:63 ← (RT)0:63

The corresponding low odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended to 64 bits then subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
- ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator EVX-form

```
evmhosmfa RT, RA, RB
```

```
| p | 4 | RT | RA | RB | 1071 |
```

RT0:31 ← (RA)16:31 ×sf (RB)16:31
RT32:63 ← (RA)48:63 ×sf (RB)48:63
ACC0:63 ← (RT)0:63

The corresponding odd-numbered, halfword signed fractional elements in RA and RB are multiplied. Each product is placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:
- ACC
Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words

evmosmfaaw RT, RA, RB

\[
\begin{array}{cccccc}
\text{temp}_0:31 & \leftarrow & (RA)_{16:31} \times_{sf} (RB)_{16:31} \\
RT_0:31 & \leftarrow & (ACC)_{0:31} + \text{temp}_0:31 \\
\text{temp}_0:31 & \leftarrow & (RA)_{48:63} \times_{sf} (RB)_{48:63} \\
RT_{32:63} & \leftarrow & (ACC)_{32:63} + \text{temp}_0:31 \\
\text{ACC}_{0:63} & \leftarrow & (RT)_{0:63}
\end{array}
\]

For each word element in the accumulator, the corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are added to the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:
- ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer

evmosmii RT, RA, RB

\[
\begin{array}{cccccc}
RT_0:31 & \leftarrow & (RA)_{16:31} \times_i (RB)_{16:31} \\
RT_{32:63} & \leftarrow & (RA)_{48:63} \times_i (RB)_{48:63}
\end{array}
\]

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
- None

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator

evmosmia RT, RA, RB

\[
\begin{array}{cccccc}
RT_0:31 & \leftarrow & (RA)_{16:31} \times_i (RB)_{16:31} \\
RT_{32:63} & \leftarrow & (RA)_{48:63} \times_i (RB)_{48:63} \\
\text{ACC}_{0:63} & \leftarrow & (RT)_{0:63}
\end{array}
\]

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:
- ACC
Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words

EVX-form

evmosmiaaw RT, RA, RB

temp\textsubscript{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}
RT\textsubscript{0:31} \leftarrow (ACC)_{0:31} + temp\textsubscript{0:31}

For each word element in the accumulator, the corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is added to the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

**Special Registers Altered:**

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words

EVX-form

evmosmianw RT, RA, RB

temp\textsubscript{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}
RT\textsubscript{0:31} \leftarrow (ACC)_{0:31} - temp\textsubscript{0:31}

For each word element in the accumulator, the corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

**Special Registers Altered:**

ACC
Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator

**EVX-form**

**EVX-form**

evhmossf RT,RA,RB

temp0:31 ← RA_{16:31} \times sf (RB)_{16:31}
if ((RA)_{16:31} = 0x8000) \& ((RB)_{16:31} = 0x8000) then
    RT_{0:31} ← 0x7FFF_FFFF
movh ← 1
else
    RT_{0:31} ← temp_{0:31}
    movh ← 0
temp0:31 ← (RA)_{48:63} \times sf (RB)_{48:63}
if ((RA)_{48:63} = 0x8000) \& ((RB)_{48:63} = 0x8000) then
    RT_{32:63} ← 0x7FFF_FFFF
movl ← 1
else
    RT_{32:63} ← temp_{0:31}
    movl ← 0
SPEFSCR\_OVH ← movh
SPEFSCR\_OV ← movl
SPEFSCR\_SOV ← SPEFSCR\_SOVH | movh
SPEFSCR\_SOV ← SPEFSCR\_SOVH | movl

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

**Special Registers Altered:**

OV OVH SOV SOVH

temp0:31 ← RA_{16:31} \times sf (RB)_{16:31}
if ((RA)_{16:31} = 0x8000) \& ((RB)_{16:31} = 0x8000) then
    RT_{0:31} ← 0x7FFF_FFFF
movh ← 1
else
    RT_{0:31} ← temp_{0:31}
    movh ← 0
temp0:31 ← (RA)_{48:63} \times sf (RB)_{48:63}
if ((RA)_{48:63} = 0x8000) \& ((RB)_{48:63} = 0x8000) then
    RT_{32:63} ← 0x7FFF_FFFF
movl ← 1
else
    RT_{32:63} ← temp_{0:31}
    movl ← 0
ACC_{0:63} ← (RT)_{0:63}
SPEFSCR\_OVH ← movh
SPEFSCR\_OV ← movl
SPEFSCR\_SOV ← SPEFSCR\_SOVH | movh
SPEFSCR\_SOV ← SPEFSCR\_SOVH | movl

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

**Special Registers Altered:**

ACC OV OVH SOV SOVH
Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words

\[ \text{evmhossfaaw RT,RA,RB} \]

\[
\begin{array}{cccccc}
\text{RT} & \text{RA} & \text{RB} & \text{1287} \\
4 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\begin{align*}
\text{temp}_{0:31} & \leftarrow (\text{RA})_{16:31} \times_\text{sf} (\text{RB})_{16:31} \\
\text{if } ((\text{RA})_{16:31} = 0x8000) \& (\text{RB})_{16:31} = 0x8000 \text{ then} \\
\text{temp}_{0:31} & \leftarrow 0x7FFF_FFFF \\
\text{movh} & \leftarrow 1 \\
\text{else} \\
\text{temp}_{0:63} & \leftarrow \text{EXTS}(\text{ACC})_{0:31} + \text{EXTS}(\text{temp}_{0:31}) \\
\text{ovh} & \leftarrow (\text{temp}_{31} \oplus \text{temp}_{32}) \\
\text{RT}_{0:31} & \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63}) \\
\text{temp}_{0:31} & \leftarrow (\text{RA})_{48:63} \times_\text{sf} (\text{RB})_{48:63} \\
\text{if } ((\text{RA})_{48:63} = 0x8000) \& (\text{RB})_{48:63} = 0x8000 \text{ then} \\
\text{temp}_{0:31} & \leftarrow 0x7FFF_FFFF \\
\text{movl} & \leftarrow 1 \\
\text{else} \\
\text{temp}_{0:63} & \leftarrow \text{EXTS}(\text{ACC})_{32:63} + \text{EXTS}(\text{temp}_{0:31}) \\
\text{ovl} & \leftarrow (\text{temp}_{31} \oplus \text{temp}_{32}) \\
\text{RT}_{32:63} & \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63}) \\
\text{ACC}_{0:63} & \leftarrow (\text{RT})_{0:63} \\
\text{SPEFSC}_\text{OVH} & \leftarrow \text{ovh} \mid \text{movh} \\
\text{SPEFSC}_\text{OV} & \leftarrow \text{ovl} \mid \text{movl} \\
\text{SPEFSC}_\text{SOVH} & \leftarrow \text{SPEFSC}_\text{SOVH} \mid \text{ovh} \mid \text{movh} \\
\text{SPEFSC}_\text{SOV} & \leftarrow \text{SPEFSC}_\text{SOV} \mid \text{ovl} \mid \text{movl}
\end{align*}
\]

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

\[ \text{ACC OV OVH SOV SOVH} \]

---

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words

\[ \text{evmhossfanw RT,RA,RB} \]

\[
\begin{array}{cccccc}
\text{RT} & \text{RA} & \text{RB} & \text{1415} \\
4 & 6 & 11 & 16 & 21 & 31 \\
\end{array}
\]

\[
\begin{align*}
\text{temp}_{0:31} & \leftarrow (\text{RA})_{16:31} \times_\text{sf} (\text{RB})_{16:31} \\
\text{if } ((\text{RA})_{16:31} = 0x8000) \& (\text{RB})_{16:31} = 0x8000 \text{ then} \\
\text{temp}_{0:31} & \leftarrow 0x7FFF_FFFF \\
\text{movh} & \leftarrow 1 \\
\text{else} \\
\text{temp}_{0:63} & \leftarrow \text{EXTS}(\text{ACC})_{0:31} - \text{EXTS}(\text{temp}_{0:31}) \\
\text{ovh} & \leftarrow (\text{temp}_{31} \oplus \text{temp}_{32}) \\
\text{RT}_{0:31} & \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63}) \\
\text{temp}_{0:31} & \leftarrow (\text{RA})_{48:63} \times_\text{sf} (\text{RB})_{48:63} \\
\text{if } ((\text{RA})_{48:63} = 0x8000) \& (\text{RB})_{48:63} = 0x8000 \text{ then} \\
\text{temp}_{0:31} & \leftarrow 0x7FFF_FFFF \\
\text{movl} & \leftarrow 1 \\
\text{else} \\
\text{temp}_{0:63} & \leftarrow \text{EXTS}(\text{ACC})_{32:63} - \text{EXTS}(\text{temp}_{0:31}) \\
\text{ovl} & \leftarrow (\text{temp}_{31} \oplus \text{temp}_{32}) \\
\text{RT}_{32:63} & \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63}) \\
\text{ACC}_{0:63} & \leftarrow (\text{RT})_{0:63} \\
\text{SPEFSC}_\text{OVH} & \leftarrow \text{ovh} \mid \text{movh} \\
\text{SPEFSC}_\text{OV} & \leftarrow \text{ovl} \mid \text{movl} \\
\text{SPEFSC}_\text{SOVH} & \leftarrow \text{SPEFSC}_\text{SOVH} \mid \text{ovh} \mid \text{movh} \\
\text{SPEFSC}_\text{SOV} & \leftarrow \text{SPEFSC}_\text{SOV} \mid \text{ovl} \mid \text{movl}
\end{align*}
\]

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

\[ \text{ACC OV OVH SOV SOVH} \]
Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words EVX-form

evmhossiaaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1285</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{temp}_{0:31} \leftarrow (\text{RA})_{16:31} \times (\text{RB})_{16:31} \]
\[ \text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{0:31}) + \text{EXTS}(\text{temp}_{0:31}) \]
\[ \text{ovh} \leftarrow (\text{temp}_{31} \oplus \text{temp}_{32}) \]
\[ \text{RT}_{0:31} \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63}) \]
\[ \text{temp}_{0:31} \leftarrow (\text{RA})_{48:63} \times (\text{RB})_{48:63} \]
\[ \text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{32:63}) + \text{EXTS}(\text{temp}_{0:31}) \]
\[ \text{ovl} \leftarrow (\text{temp}_{31} \oplus \text{temp}_{32}) \]
\[ \text{RT}_{32:63} \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_{31}, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63}) \]
\[ \text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \]

SPEFSCROVH \leftarrow ovh
SPEFSCROV \leftarrow ovl
SPEFSCRSOVH \leftarrow SPEFSCRSOVH \leftarrow ovh
SPEFSCRSOV \leftarrow SPEFSCRSOV \leftarrow ovl

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer EVX-form
evmhoumi RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1036</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{RT}_{0:31} \leftarrow (\text{RA})_{16:31} \times_{ui} (\text{RB})_{16:31} \]
\[ \text{RT}_{32:63} \leftarrow (\text{RA})_{48:63} \times_{ui} (\text{RB})_{48:63} \]

The corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator EVX-form
evmhoumia RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1068</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{RT}_{0:31} \leftarrow (\text{RA})_{16:31} \times_{ui} (\text{RB})_{16:31} \]
\[ \text{RT}_{32:63} \leftarrow (\text{RA})_{48:63} \times_{ui} (\text{RB})_{48:63} \]
\[ \text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \]

The corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into RT and into the accumulator.

Special Registers Altered:
ACC
Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words \textit{EVX-form}

\texttt{evmhoumiaaw RT,RA,RB}

\begin{tabular}{cccccc}
  p & 4 & RT & RA & RB & 1292 \hline
  0 & 6 & 11 & 16 & 21 & 31
\end{tabular}

\begin{align*}
  \text{temp}_{0:32} & \leftarrow (RA)_{16:31} \times \text{ui} (RB)_{16:31} \\
  \text{RT}_{0:31} & \leftarrow (\text{ACC})_{0:31} + \text{temp}_{0:32} \\
  \text{temp}_{0:32} & \leftarrow (RA)_{48:63} \times \text{ui} (RB)_{48:63} \\
  \text{RT}_{32:63} & \leftarrow (\text{ACC})_{32:63} + \text{temp}_{0:31} \\
  \text{ACC}_{0:63} & \leftarrow (\text{RT})_{0:63}
\end{align*}

For each word element in the accumulator, the corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding RT and accumulator words.

Special Registers Altered:

\texttt{ACC}

---

Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words \textit{EVX-form}

\texttt{evmhousiaaw RT,RA,RB}

\begin{tabular}{cccccc}
  p & 4 & RT & RA & RB & 1284 \hline
  0 & 6 & 11 & 16 & 21 & 31
\end{tabular}

\begin{align*}
  \text{temp}_{0:32} & \leftarrow (RA)_{16:31} \times \text{ui} (RB)_{16:31} \\
  \text{temp}_{0:63} & \leftarrow \text{EXTZ}((\text{ACC})_{0:32}) + \text{EXTZ}(\text{temp}_{0:32}) \\
  \text{ovh} & \leftarrow \text{temp}_{31} \\
  \text{RT}_{0:31} & \leftarrow \text{SATURATE}(\text{ovh}, 0, 0x0000\_FFFF, 0x0000\_FFFF, \text{temp}_{32:63}) \\
  \text{temp}_{0:32} & \leftarrow (RA)_{48:63} \times \text{ui} (RB)_{48:63} \\
  \text{temp}_{0:63} & \leftarrow \text{EXTZ}((\text{ACC})_{32:63}) + \text{EXTZ}(\text{temp}_{0:31}) \\
  \text{ovl} & \leftarrow \text{temp}_{31} \\
  \text{RT}_{32:63} & \leftarrow \text{SATURATE}(\text{ovl}, 0, 0x0000\_FFFF, 0x0000\_FFFF, \text{temp}_{32:63}) \\
  \text{ACC}_{0:63} & \leftarrow (\text{RT})_{0:63} \\
  \text{SPEFSCROvh} & \leftarrow \text{ovh} \\
  \text{SPEFSCROv} & \leftarrow \text{ovl} \\
  \text{SPEFSCROvh} & \leftarrow \text{SPEFSCROvh} \mid \text{ovh} \\
  \text{SPEFSCROv} & \leftarrow \text{SPEFSCROv} \mid \text{ovl}
\end{align*}

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

\texttt{ACC OV OVH SOV SOVH}

---

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words \textit{EVX-form}

\texttt{evmhoumianw RT,RA,RB}

\begin{tabular}{cccccc}
  p & 4 & RT & RA & RB & 1420 \hline
  0 & 6 & 11 & 16 & 21 & 31
\end{tabular}

\begin{align*}
  \text{temp}_{0:31} & \leftarrow (RA)_{16:31} \times \text{ui} (RB)_{16:31} \\
  \text{temp}_{0:63} & \leftarrow \text{EXTZ}((\text{ACC})_{0:31}) - \text{EXTZ}(\text{temp}_{0:31}) \\
  \text{ovh} & \leftarrow \text{temp}_{31} \\
  \text{RT}_{0:31} & \leftarrow \text{SATURATE}(\text{ovh}, 0, 0x0000\_0000, 0x0000\_0000, \text{temp}_{32:63}) \\
  \text{temp}_{0:31} & \leftarrow (RA)_{48:63} \times \text{ui} (RB)_{48:63} \\
  \text{temp}_{0:63} & \leftarrow \text{EXTZ}((\text{ACC})_{32:63}) - \text{EXTZ}(\text{temp}_{0:31}) \\
  \text{ovl} & \leftarrow \text{temp}_{31} \\
  \text{RT}_{32:63} & \leftarrow \text{SATURATE}(\text{ovl}, 0, 0x0000\_0000, 0x0000\_0000, \text{temp}_{32:63}) \\
  \text{ACC}_{0:63} & \leftarrow (\text{RT})_{0:63} \\
  \text{SPEFSCROvh} & \leftarrow \text{ovh} \\
  \text{SPEFSCROv} & \leftarrow \text{ovl} \\
  \text{SPEFSCROvh} & \leftarrow \text{SPEFSCROvh} \mid \text{ovh} \\
  \text{SPEFSCROv} & \leftarrow \text{SPEFSCROv} \mid \text{ovl}
\end{align*}

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

\texttt{ACC OV OVH SOV SOVH}

---

Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words \textit{EVX-form}

\texttt{evmhousianw RT,RA,RB}

\begin{tabular}{cccccc}
  p & 4 & RT & RA & RB & 1412 \hline
  0 & 6 & 11 & 16 & 21 & 31
\end{tabular}

\begin{align*}
  \text{temp}_{0:31} & \leftarrow (RA)_{16:31} \times \text{ui} (RB)_{16:31} \\
  \text{temp}_{0:63} & \leftarrow \text{EXTZ}((\text{ACC})_{0:31}) - \text{EXTZ}(\text{temp}_{0:31}) \\
  \text{ovh} & \leftarrow \text{temp}_{31} \\
  \text{RT}_{0:31} & \leftarrow \text{SATURATE}(\text{ovh}, 0, 0x0000\_0000, 0x0000\_0000, \text{temp}_{32:63}) \\
  \text{temp}_{0:31} & \leftarrow (RA)_{48:63} \times \text{ui} (RB)_{48:63} \\
  \text{temp}_{0:63} & \leftarrow \text{EXTZ}((\text{ACC})_{32:63}) - \text{EXTZ}(\text{temp}_{0:31}) \\
  \text{ovl} & \leftarrow \text{temp}_{31} \\
  \text{RT}_{32:63} & \leftarrow \text{SATURATE}(\text{ovl}, 0, 0x0000\_0000, 0x0000\_0000, \text{temp}_{32:63}) \\
  \text{ACC}_{0:63} & \leftarrow (\text{RT})_{0:63} \\
  \text{SPEFSCROvh} & \leftarrow \text{ovh} \\
  \text{SPEFSCROv} & \leftarrow \text{ovl} \\
  \text{SPEFSCROvh} & \leftarrow \text{SPEFSCROvh} \mid \text{ovh} \\
  \text{SPEFSCROv} & \leftarrow \text{SPEFSCROv} \mid \text{ovl}
\end{align*}

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

\texttt{ACC OV OVH SOV SOVH}
**Initialize Accumulator**

*EVX-form*

\[
\text{evmra} \quad \text{RT}, \text{RA}
\]

The contents of RA are placed into the accumulator and RT. This is the method for initializing the accumulator.

**Special Registers Altered:**

- ACC

**Vector Multiply Word High Signed, Modulo, Fractional**

*EVX-form*

\[
\text{evmwhsmf} \quad \text{RT}, \text{RA}, \text{RB}
\]

The corresponding word signed fractional elements in RA and RB are multiplied and bits 0:31 of the two products are placed into the two corresponding words of RT.

**Special Registers Altered:**

- None

**Vector Multiply Word High Signed, Modulo, Integer**

*EVX-form*

\[
\text{evmwhsmi} \quad \text{RT}, \text{RA}, \text{RB}
\]

The corresponding word signed-integer elements in RA and RB are multiplied. Bits 0:31 of the two 64-bit products are placed into the two corresponding words of RT.

**Special Registers Altered:**

- None

**Vector Multiply Word High Signed, Modulo, Integer to Accumulator**

*EVX-form*

\[
\text{evmwhsmia} \quad \text{RT}, \text{RA}, \text{RB}
\]

The corresponding word signed-integer elements in RA and RB are multiplied. Bits 0:31 of the two 64-bit products are placed into the two corresponding words of RT and into the accumulator.

**Special Registers Altered:**

- ACC
Vector Multiply Word High Signed, Saturate, Fractional
EVX-form

`evmwhssf` RT,RA,RB

```
4  6 11 16 21 31
```

```
temp0:63 ← (RA)0:31 ×sf (RB)0:31
if ((RA)0:31 = 0x8000_0000) & ((RB)0:31 = 0x8000_0000)
then
  RT0:31 ← 0x7FFF_FFFF
  movh ← 1
else
  RT0:31 ← temp0:31
  movh ← 0
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63 = 0x8000_0000) & ((RB)32:63 = 0x8000_0000)
then
  RT32:63 ← 0x7FFF_FFFF
  movl ← 1
else
  RT32:63 ← temp0:31
  movl ← 0
SPEFSCROV ← movh
SPEFSCROV ← movl
SPEFSCROVH ← SPEFSCROV | movh
SPEFSCROVH ← SPEFSCROV | movl
```

The corresponding word signed fractional elements in RA and RB are multiplied. Bits 0:31 of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:
OV OVH SOV SOVH

Vector Multiply Word High Unsigned, Modulo, Integer
EVX-form

`evmwhumi` RT,RA,RB

```
4  6 11 16 21 31
```

```
temp0:63 ← (RA)0:31 ×ui (RB)0:31
RT0:31 ← temp0:31
temp0:63 ← (RA)32:63 ×ui (RB)32:63
RT32:63 ← temp0:31
```

The corresponding word unsigned-integer elements in RA and RB are multiplied. Bits 0:31 of the two products are placed into the two corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX-form

`evmwhumia` RT,RA,RB

```
4  6 11 16 21 31
```

```
temp0:63 ← (RA)0:31 ×ui (RB)0:31
RT0:31 ← temp0:31
temp0:63 ← (RA)32:63 ×ui (RB)32:63
RT32:63 ← temp0:31
ACC0:63 ← (RT)0:63
```

The corresponding word unsigned-integer elements in RA and RB are multiplied. Bits 0:31 of the two products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words EVX-form

evmlsmiaaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1353</th>
</tr>
</thead>
</table>

$\text{temp}_{0:63} \leftarrow (\text{RA})_{0:31} \times (\text{RB})_{0:31}$
$\text{RT}_{0:31} \leftarrow (\text{ACC})_{0:31} + \text{temp}_{32:63}$
$\text{temp}_{0:63} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}$
$\text{RT}_{32:63} \leftarrow (\text{ACC})_{32:63} + \text{temp}_{32:63}$
$\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}$

For each word element in the accumulator, the corresponding word signed-integer elements in RA and RB are multiplied. The least significant 32 bits of each intermediate product are added to the contents of the corresponding accumulator words, and the result is placed in RT and the accumulator.

**Special Registers Altered:**
- $\text{ACC}$

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words EVX-form

evmlssiaaw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1345</th>
</tr>
</thead>
</table>

$\text{temp}_{0:63} \leftarrow (\text{RA})_{0:31} \times (\text{RB})_{0:31}$
$\text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{0:31}) + \text{EXTS}(\text{temp}_{32:63})$
$\text{ovh} \leftarrow (\text{temp}_31 \oplus \text{temp}_32)$
$\text{RT}_{0:31} \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_31, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})$
$\text{temp}_{0:63} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}$
$\text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{32:63}) + \text{EXTS}(\text{temp}_{32:63})$
$\text{ovl} \leftarrow (\text{temp}_31 \oplus \text{temp}_32)$
$\text{RT}_{32:63} \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_31, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})$
$\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}$
$\text{SPEFSCR}_\text{OVH} \leftarrow \text{ovh}$
$\text{SPEFSCR}_\text{OV} \leftarrow \text{ovl}$
$\text{SPEFSCR}_\text{SOV} \leftarrow \text{SPEFSCR}_\text{SOV} \mid \text{ovh}$
$\text{SPEFSCR}_\text{SOV} \leftarrow \text{SPEFSCR}_\text{SOV} \mid \text{ovl}$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

**Special Registers Altered:**
- $\text{ACC OV OVH SOV SOVH}$

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words EVX-form
evmlsmianw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1481</th>
</tr>
</thead>
</table>

$\text{temp}_{0:63} \leftarrow (\text{RA})_{0:31} \times (\text{RB})_{0:31}$
$\text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{0:31}) - \text{EXTS}(\text{temp}_{32:63})$
$\text{ovh} \leftarrow (\text{temp}_31 \oplus \text{temp}_32)$
$\text{RT}_{0:31} \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_31, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})$
$\text{temp}_{0:63} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}$
$\text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{32:63}) - \text{EXTS}(\text{temp}_{32:63})$
$\text{ovl} \leftarrow (\text{temp}_31 \oplus \text{temp}_32)$
$\text{RT}_{32:63} \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_31, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})$
$\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}$
$\text{SPEFSCR}_\text{OVH} \leftarrow \text{ovh}$
$\text{SPEFSCR}_\text{OV} \leftarrow \text{ovl}$
$\text{SPEFSCR}_\text{SOV} \leftarrow \text{SPEFSCR}_\text{SOV} \mid \text{ovh}$
$\text{SPEFSCR}_\text{SOV} \leftarrow \text{SPEFSCR}_\text{SOV} \mid \text{ovl}$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the contents of the corresponding accumulator words and the result is placed in RT and the accumulator.

**Special Registers Altered:**
- $\text{ACC OV OVH SOV SOVH}$

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words EVX-form
evmlssianw RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1473</th>
</tr>
</thead>
</table>

$\text{temp}_{0:63} \leftarrow (\text{RA})_{0:31} \times (\text{RB})_{0:31}$
$\text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{0:31}) - \text{EXTS}(\text{temp}_{32:63})$
$\text{ovh} \leftarrow (\text{temp}_31 \oplus \text{temp}_32)$
$\text{RT}_{0:31} \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_31, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})$
$\text{temp}_{0:63} \leftarrow (\text{RA})_{32:63} \times (\text{RB})_{32:63}$
$\text{temp}_{0:63} \leftarrow \text{EXTS}((\text{ACC})_{32:63}) - \text{EXTS}(\text{temp}_{32:63})$
$\text{ovl} \leftarrow (\text{temp}_31 \oplus \text{temp}_32)$
$\text{RT}_{32:63} \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_31, 0x8000_0000, 0x7FFF_FFFF, \text{temp}_{32:63})$
$\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}$
$\text{SPEFSCR}_\text{OVH} \leftarrow \text{ovh}$
$\text{SPEFSCR}_\text{OV} \leftarrow \text{ovl}$
$\text{SPEFSCR}_\text{SOV} \leftarrow \text{SPEFSCR}_\text{SOV} \mid \text{ovh}$
$\text{SPEFSCR}_\text{SOV} \leftarrow \text{SPEFSCR}_\text{SOV} \mid \text{ovl}$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

**Special Registers Altered:**
- $\text{ACC OV OVH SOV SOVH}$

Chapter 8. Signal Processing Engine (SPE) 625
Vector Multiply Word Low Unsigned, Modulo, Integer

```
evmlumi RT,RA,RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1096</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}
\]
\[
RT_{0:31} \leftarrow temp_{32:63}
\]
\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}
\]
\[
RT_{32:63} \leftarrow temp_{32:63}
\]

The corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of RT.

Special Registers Altered:

- None

- Programming Note

The least significant 32 bits of the product are independent of whether the word elements in RA and RB are treated as signed or unsigned 32-bit integers.

Note that `evmwlumi` can be used for signed or unsigned integers.

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words

```
evmwlumiaaw RT,RA,RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1352</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}
\]
\[
RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{32:63}
\]
\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}
\]
\[
RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{32:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

For each word element in the accumulator, the corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are added to the contents of the corresponding accumulator word and the result is placed in RT and the accumulator.

Special Registers Altered:

- ACC

Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator

```
evmwlumia RT,RA,RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1128</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}
\]
\[
RT_{0:31} \leftarrow temp_{32:63}
\]
\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}
\]
\[
RT_{32:63} \leftarrow temp_{32:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:

- ACC

- Programming Note

The least significant 32 bits of the product are independent of whether the word elements in RA and RB are treated as signed or unsigned 32-bit integers.

Note that `evmwlumia` can be used for signed or unsigned integers.

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words

```
evmwlumianw RT,RA,RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1480</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}
\]
\[
RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{32:63}
\]
\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}
\]
\[
RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{32:63}
\]
\[
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

For each word element in the accumulator, the corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are subtracted from the contents of the corresponding accumulator word and the result is placed in RT and the accumulator.

Special Registers Altered:

- ACC
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words EVX-form

evmlusiaaw RT, RA, RB

For each word element in the accumulator, corresponding word unsigned-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Multiply Word Signed, Modulo, Fractional EVX-form

evmsmf RT, RA, RB

The corresponding low word signed fractional elements in RA and RB are multiplied. The product is placed in RT.

Special Registers Altered:
None
Vector Multiply Word Signed, Modulo, Fractional and Accumulate

\[ \text{evmwsmfaa} \quad RT, RA, RB \]

\[
\begin{array}{cccccc}
 4 & 6 & 8 & 10 & 11 & 1371 \\
\end{array}
\]

\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{af} (RB)_{32:63} \\
RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63} \\
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The corresponding low word signed fractional elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Signed, Modulo, Integer

\[ \text{evmwsmi} \quad RT, RA, RB \]

\[
\begin{array}{cccccc}
 4 & 6 & 8 & 10 & 11 & 1113 \\
\end{array}
\]

\[
RT_{0:63} \leftarrow (RA)_{32:63} \times_{ai} (RB)_{32:63} \\
\]

The low word signed-integer elements in RA and RB are multiplied. The product is placed in RT.

Special Registers Altered:

None

Vector Multiply Word Signed, Modulo, Integer and Accumulate

\[ \text{evmwsmiaa} \quad RT, RA, RB \]

\[
\begin{array}{cccccc}
 4 & 6 & 8 & 10 & 11 & 1369 \\
\end{array}
\]

\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{ai} (RB)_{32:63} \\
RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63} \\
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The low word signed-integer elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative

\[ \text{evmwsmian} \quad RT, RA, RB \]

\[
\begin{array}{cccccc}
 4 & 6 & 8 & 10 & 11 & 1497 \\
\end{array}
\]

\[
temp_{0:63} \leftarrow (RA)_{32:63} \times_{ai} (RB)_{32:63} \\
RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63} \\
ACC_{0:63} \leftarrow (RT)_{0:63}
\]

The low word signed-integer elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC
Vector Multiply Word Signed, Saturate, Fractional, EVX-form

evmwssf RT, RA, RB

\[
\begin{array}{cccccc}
4 & RT & RA & RB & 1107 & 31 \\
\end{array}
\]

\[
temp_{0.63} \leftarrow (RA)_{32.63} \times sf (RB)_{32.63}
\]

if ((RA)_{32.63} = 0x8000_0000) \& (RB)_{32.63} = 0x8000_0000
then
\[
RT_{0.63} \leftarrow 0x7FFFF_FFFF_FFFF_FFFF
\]
mov \leftarrow 1
else
\[
RT_{0.63} \leftarrow temp_{0.63}
\]
mov \leftarrow 0
SPEFSR_OVH \leftarrow 0
SPEFSR_OV \leftarrow mov
SPEFSR_OVH \leftarrow SPEFSR_OV | mov

The low word signed fractional elements in RA and RB are multiplied. The 64-bit product is placed in RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional to Accumulator, EVX-form

evmwssfa RT, RA, RB

\[
\begin{array}{cccccc}
4 & RT & RA & RB & 1139 & 31 \\
\end{array}
\]

\[
temp_{0.63} \leftarrow (RA)_{32.63} \times sf (RB)_{32.63}
\]

if ((RA)_{32.63} = 0x8000_0000) \& (RB)_{32.63} = 0x8000_0000
then
\[
RT_{0.63} \leftarrow 0x7FFFF_FFFF_FFFF_FFFF
\]
mov \leftarrow 1
else
\[
RT_{0.63} \leftarrow temp_{0.63}
\]
mov \leftarrow 0
ACCO_{0.63} \leftarrow (RT)_{0.63}
SPEFSR_OVH \leftarrow 0
SPEFSR_OV \leftarrow mov
SPEFSR_OVH \leftarrow SPEFSR_OV | mov

The low word signed fractional elements in RA and RB are multiplied. The 64-bit product is placed in RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

ACC OV OVH SOV
Vector Multiply Word Signed, Saturate, Fractional and Accumulate EVX-form

```
evmwssfaa RT,RA,RB
```

```
\text{temp}_0:63 \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}
\text{if } ((RA)_{32:63}=0x8000\_0000) \& ((RB)_{32:63}=0x8000\_0000)
\text{then}
\quad \text{mov} \leftarrow 1
\text{else}
\quad \text{mov} \leftarrow 0
\text{temp}_0:64 \leftarrow \text{EXTS}((\text{ACC})_{0:63}) + \text{EXTS}(\text{temp}_0:63)
\text{ov} \leftarrow (\text{temp}_0 \oplus \text{temp}_1)
\text{RT}_0:63 \leftarrow \text{temp}_1:64
\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}
\text{SPEFSCR}_{OVH} \leftarrow 0
\text{SPEFSCR}_{OV} \leftarrow \text{ov} \mid \text{mov}
\text{SPEFSCR}_{SOV} \leftarrow \text{SPEFSCR}_{SOV} \mid \text{ov} \mid \text{mov}
```

The low word signed fractional elements in RA and RB are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then added to the accumulator and the result is placed in RT and the accumulator.

**Special Registers Altered:**

- ACC
- OV
- OVH
- SOV

Vector Multiply Word Unsigned, Modulo, Integer EVX-form

```
evmwumi RT,RA,RB
```

```
\text{RT}_0:63 \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}
The low word unsigned-integer elements in RA and RB are multiplied to form a 64-bit product that is placed in RT.
```

**Special Registers Altered:**

None

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative EVX-form

```
evmwssfan RT,RA,RB
```

```
\text{temp}_0:63 \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}
\text{if } ((RA)_{32:63}=0x8000\_0000) \& ((RB)_{32:63}=0x8000\_0000)
\text{then}
\quad \text{mov} \leftarrow 1
\text{else}
\quad \text{mov} \leftarrow 0
\text{temp}_0:64 \leftarrow \text{EXTS}((\text{ACC})_{0:63}) - \text{EXTS}(\text{temp}_0:63)
\text{ov} \leftarrow (\text{temp}_0 \oplus \text{temp}_1)
\text{RT}_0:63 \leftarrow \text{temp}_1:64
\text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63}
\text{SPEFSCR}_{OVH} \leftarrow 0
\text{SPEFSCR}_{OV} \leftarrow \text{ov} \mid \text{mov}
\text{SPEFSCR}_{SOV} \leftarrow \text{SPEFSCR}_{SOV} \mid \text{ov} \mid \text{mov}
```

The low word signed fractional elements in RA and RB are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then subtracted from the accumulator and the result is placed in RT and the accumulator.

**Special Registers Altered:**

- ACC
- OV
- OVH
- SOV

Vector Multiply Word Unsigned, Modulo, Integer to Accumulator EVX-form

```
evmwumia RT,RA,RB
```

```
\text{RT}_0:63 \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}
The low word unsigned-integer elements in RA and RB are multiplied to form a 64-bit product that is placed in RT.
```

**Special Registers Altered:**

- ACC
**Vector Multiply Word Unsigned, Modulo, Integer and Accumulate** EVX-form

evmwumia RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>1368</th>
</tr>
</thead>
</table>

| temp0:63 | $\left[ \text{RA} \right]_{32:63} \times_{u1} \left[ \text{RB} \right]_{32:63}$ |
| RT0:63 | $\left( \text{ACC} \right)_{0:63} + \text{temp0:63}$ |
| ACC0:63 | $\left( \text{RT} \right)_{0:63}$ |

The low word unsigned-integer elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and in RT.

**Special Registers Altered:**

ACC

**Vector NAND** EVX-form

evnand RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>542</th>
</tr>
</thead>
</table>

| RT0:31 | $\neg \left( \left[ \text{RA} \right]_{0:31} \& \left[ \text{RB} \right]_{0:31} \right)$ |
| RT32:63 | $\neg \left( \left[ \text{RA} \right]_{32:63} \& \left[ \text{RB} \right]_{32:63} \right)$ |

Each element of RA and RB is bitwise NANDed. The result is placed in the corresponding element of RT.

**Special Registers Altered:**

None

**Vector Negate** EVX-form

evneg RT,RA

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>521</th>
</tr>
</thead>
</table>

| RT0:31 | $\text{NEG} \left( \left[ \text{RA} \right]_{0:31} \right)$ |
| RT32:63 | $\text{NEG} \left( \left[ \text{RA} \right]_{32:63} \right)$ |

The negative of each element of RA is placed in RT. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000.

**Special Registers Altered:**

None
### Vector NOR

**EVX-form**

```
evnor RT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>31</th>
<th>32</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
RT_{0:31} & \leftarrow \neg ((RA)_{0:31} \mid (RB)_{0:31}) \\
RT_{32:63} & \leftarrow \neg ((RA)_{32:63} \mid (RB)_{32:63})
\end{align*}
\]

Each element of RA and RB is bitwise NORed. The result is placed in the corresponding element of RT.

**Special Registers Altered:** None

**Extended Mnemonics:**

Extended mnemonics are provided for the Vector NOR instruction to produce a vector bitwise complement operation.

**Extended:** Equivalent to:

```
evnot RT,RA
evnor RT,RA,RA
```

### Vector OR

**EVX-form**

```
evor RT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>31</th>
<th>32</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
RT_{0:31} & \leftarrow (RA)_{0:31} \mid (RB)_{0:31} \\
RT_{32:63} & \leftarrow (RA)_{32:63} \mid (RB)_{32:63}
\end{align*}
\]

Each element of RA and RB is bitwise ORed. The result is placed in the corresponding element of RT.

**Special Registers Altered:** None

**Extended Mnemonics:**

Extended mnemonics are provided for the Vector OR instruction to provide a 64-bit vector move instruction.

**Extended:** Equivalent to:

```
evmr RT,RA
evor RT,RA,RA
```

### Vector OR with Complement

**EVX-form**

```
evorc RT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>31</th>
<th>32</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
RT_{0:31} & \leftarrow (RA)_{0:31} \mid \neg (RB)_{0:31} \\
RT_{32:63} & \leftarrow (RA)_{32:63} \mid \neg (RB)_{32:63}
\end{align*}
\]

Each element of RA is bitwise ORed with the complement of RB. The result is placed in the corresponding element of RT.

**Special Registers Altered:** None

### Vector Rotate Left Word

**EVX-form**

```
evrlw RT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>31</th>
<th>32</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{nh} & \leftarrow (RB)_{27:31} \\
\text{nl} & \leftarrow (RB)_{59:63} \\
RT_{0:31} & \leftarrow \text{ROTL}((RA)_{0:31}, \text{nh}) \\
RT_{32:63} & \leftarrow \text{ROTL}((RA)_{32:63}, \text{nl})
\end{align*}
\]

Each of the high and low elements of RA is rotated left by an amount specified in RB. The result is placed in RT. Rotate values for each element of RA are found in bit positions RB_{27:31} and RB_{59:63}.

**Special Registers Altered:** None
**Vector Rotate Left Word Immediate**

*EVX-form*

```
evrwi    RT,RA,UI
```

$$\begin{array}{cc}
\text{4} & \text{RT} & \text{RA} & \text{UI} & \text{554} \\
\end{array}$$

\[n \leftarrow \text{UI}\]
\[\text{RT}_{0:31} \leftarrow \text{ROTL}((\text{RA})_{0:31}, n)\]
\[\text{RT}_{32:63} \leftarrow \text{ROTL}((\text{RA})_{32:63}, n)\]

Both the high and low elements of RA are rotated left by an amount specified by UI.

**Special Registers Altered:**

None

---

**Vector Round Word**

*EVX-form*

```
evrdw    RT,RA
```

$$\begin{array}{cc}
\text{4} & \text{RT} & \text{RA} & /// & \text{524} \\
\end{array}$$

\[\text{RT}_{0:31} \leftarrow ((\text{RA})_{0:31} + 0x00008000) \& 0xFFFF0000\]
\[\text{RT}_{32:63} \leftarrow ((\text{RA})_{32:63} + 0x00008000) \& 0xFFFF0000\]

The 32-bit elements of RA are rounded into 16 bits. The result is placed in RT. The resulting 16 bits are placed in the most significant 16 bits of each element of RT, zeroing out the low-order 16 bits of each element.

**Special Registers Altered:**

None

---

**Vector Select**

*EVS-form*

```
evsel    RT,RA,RB,BFA
```

$$\begin{array}{cc}
\text{4} & \text{RT} & \text{RA} & \text{RB} & \text{BFA} & \text{79} \\
\end{array}$$

\[\text{ch} \leftarrow \text{CR}_{\text{BFA}\_4}\]
\[\text{cl} \leftarrow \text{CR}_{\text{BFA}\_4+1}\]

if (ch = 1) then
\[\text{RT}_{0:31} \leftarrow (\text{RA})_{0:31}\]
else
\[\text{RT}_{0:31} \leftarrow (\text{RB})_{0:31}\]

if (cl = 1) then
\[\text{RT}_{32:63} \leftarrow (\text{RA})_{32:63}\]
else
\[\text{RT}_{32:63} \leftarrow (\text{RB})_{32:63}\]

If the most significant bit in the BFA field of CR is set to 1, the high-order element of RA is placed in the high-order element of RT; otherwise, the high-order element of RB is placed into the high-order element of RT. If the next most significant bit in the BFA field of CR is set to 1, the low-order element of RA is placed in the low-order element of RT, otherwise, the low-order element of RB is placed into the low-order element of RT.

**Special Registers Altered:**

None
Vector Shift Left Word

**EVX-form**

```
evslw RT, RA, RB
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>548</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ \text{nh} \leftarrow (RB)_{26:31} \]
\[ \text{nl} \leftarrow (RB)_{58:63} \]
\[ RT_{0:31} \leftarrow \text{SL}((RA)_{0:31}, \text{nh}) \]
\[ RT_{32:63} \leftarrow \text{SL}((RA)_{32:63}, \text{nl}) \]

Each of the high and low elements of RA is shifted left by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63.

Shift amounts from 32 to 63 give a zero result.

**Special Registers Altered:**

None

Vector Shift Left Word Immediate

**EVX-form**

```
evslwi RT, RA, UI
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>550</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ n \leftarrow \text{UI} \]
\[ RT_{0:31} \leftarrow \text{SL}((RA)_{0:31}, n) \]
\[ RT_{32:63} \leftarrow \text{SL}((RA)_{32:63}, n) \]

Both high and low elements of RA are shifted left by the 5-bit UI value and the results are placed in RT.

**Special Registers Altered:**

None

Vector Splat Fractional Immediate

**EVX-form**

```
evslafi RT, SI
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>SI</th>
<th>///</th>
<th>555</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ RT_{0:31} \leftarrow \text{SI} || {270} \]
\[ RT_{32:63} \leftarrow \text{SI} || {270} \]

The value specified by SI is padded with trailing zeros and placed in both elements of RT. The SI ends up in bit positions RT_{0:4} and RT_{32:36}.

**Special Registers Altered:**

None

Vector Splat Immediate

**EVX-form**

```
evsplati RT, SI
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>SI</th>
<th>///</th>
<th>553</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ RT_{0:31} \leftarrow \text{EXTS(SI)} \]
\[ RT_{32:63} \leftarrow \text{EXTS(SI)} \]

The value specified by SI is sign extended and placed in both elements of RT.

**Special Registers Altered:**

None

Vector Shift Right Word Immediate Signed

**EVX-form**

```
evsrwis RT, RA, UI
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>547</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ n \leftarrow \text{UI} \]
\[ RT_{0:31} \leftarrow \text{EXTS}((RA)_{0:31-n}) \]
\[ RT_{32:63} \leftarrow \text{EXTS}((RA)_{32:63-n}) \]

Both high and low elements of RA are shifted right by the 5-bit UI value. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

**Special Registers Altered:**

None

Vector Shift Right Word Immediate Unsigned

**EVX-form**

```
evswiui RT, RA, UI
```

<table>
<thead>
<tr>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>UI</th>
<th>546</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

\[ n \leftarrow \text{UI} \]
\[ RT_{0:31} \leftarrow \text{EXTZ}((RA)_{0:31-n}) \]
\[ RT_{32:63} \leftarrow \text{EXTZ}((RA)_{32:63-n}) \]

Both high and low elements of RA are shifted right by the 5-bit UI value; zeros are shifted into the most significant position.

**Special Registers Altered:**

None
**Vector Shift Right Word Signed EVX-form**

```
vector_vector_shift_right_word_signed

```evsrws


RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>545</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nh ← (RB)_{26:31}
nl ← (RB)_{58:63}
RT_{0:31} ← EXTS((RA)_{0:31}-nh)
RT_{32:63} ← EXTS((RA)_{32:63}-nl)

Both the high and low elements of RA are shifted right by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63. The sign bits are shifted into the most significant position.

Shift amounts from 32 to 63 give a result of 32 sign bits.

**Special Registers Altered:**
None

**Vector Shift Right Word Unsigned EVX-form**

```
vector_vector_shift_right_word_unsigned

```evsrwu


RT,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>544</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nh ← (RB)_{26:31}
nl ← (RB)_{58:63}
RT_{0:31} ← EXTZ((RA)_{0:31}-nh)
RT_{32:63} ← EXTZ((RA)_{32:63}-nl)

Both the high and low elements of RA are shifted right by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63. Zeros are shifted into the most significant position.

Shift amounts from 32 to 63 give a zero result.

**Special Registers Altered:**
None

**Vector Store Double of Double EVX-form**

```
vector_store_double_of_double

```evstdd


RS,D(RA)

<table>
<thead>
<tr>
<th>4</th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
<th>801</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI × 8)
MEM(EA, 8) ← (RS)_{0:63}

D in the instruction mnemonic is UI × 8. The contents of RS are stored as a doubleword in storage addressed by EA.

**Special Registers Altered:**
None

**Vector Store Double of Double Indexed EVX-form**

```
vector_store_double_of_double_indexed

```evstddx


RS,RA,RB

<table>
<thead>
<tr>
<th>4</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)_{0:63}

The contents of RS are stored as a doubleword in storage addressed by EA.

**Special Registers Altered:**
None
### Vector Store Double of Four Halfwords
#### EVX-form

**evstdh RS,D(RA)**

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>805</td>
</tr>
</tbody>
</table>

- if \((RA = 0)\) then \(b \leftarrow 0\)
- else \(b \leftarrow (RA)\)

\[
\begin{align*}
EA & \leftarrow b + \text{EXT2}(UI \times 8) \\
\text{MEM}(EA,2) & \leftarrow (RS)_{0:15} \\
\text{MEM}(EA+2,2) & \leftarrow (RS)_{16:31} \\
\text{MEM}(EA+4,2) & \leftarrow (RS)_{32:47} \\
\text{MEM}(EA+6,2) & \leftarrow (RS)_{48:63}
\end{align*}
\]

D in the instruction mnemonic is \(UI \times 8\). The contents of RS are stored as four halfwords in storage addressed by EA.

**Special Registers Altered:** None

#### Indexed

**evstdhx RS,RA,RB**

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>804</td>
</tr>
</tbody>
</table>

- if \((RA = 0)\) then \(b \leftarrow 0\)
- else \(b \leftarrow (RA)\)

\[
\begin{align*}
EA & \leftarrow b + (RB) \\
\text{MEM}(EA,2) & \leftarrow (RS)_{0:15} \\
\text{MEM}(EA+2,2) & \leftarrow (RS)_{16:31} \\
\text{MEM}(EA+4,2) & \leftarrow (RS)_{32:47} \\
\text{MEM}(EA+6,2) & \leftarrow (RS)_{48:63}
\end{align*}
\]

The contents of RS are stored as four halfwords in storage addressed by EA.

**Special Registers Altered:** None

### Vector Store Double of Two Words
#### EVX-form

**evstdw RS,D(RA)**

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>803</td>
</tr>
</tbody>
</table>

- if \((RA = 0)\) then \(b \leftarrow 0\)
- else \(b \leftarrow (RA)\)

\[
\begin{align*}
EA & \leftarrow b + \text{EXT2}(UI \times 8) \\
\text{MEM}(EA,4) & \leftarrow (RS)_{0:31} \\
\text{MEM}(EA+4,4) & \leftarrow (RS)_{32:63}
\end{align*}
\]

D in the instruction mnemonic is \(UI \times 8\). The contents of RS are stored as two words in storage addressed by EA.

**Special Registers Altered:** None

#### Indexed

**evstdwx RS,RA,RB**

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>802</td>
</tr>
</tbody>
</table>

- if \((RA = 0)\) then \(b \leftarrow 0\)
- else \(b \leftarrow (RA)\)

\[
\begin{align*}
EA & \leftarrow b + (RB) \\
\text{MEM}(EA,4) & \leftarrow (RS)_{0:31} \\
\text{MEM}(EA+4,4) & \leftarrow (RS)_{32:63}
\end{align*}
\]

The contents of RS are stored as two words in storage addressed by EA.

**Special Registers Altered:** None
**Vector Store Word of Two Halfwords from Even EVX-form**

```
evstwhe RS,D(RA)
```

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
<th>817</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then $b \leftarrow 0$
else $b \leftarrow (RA)$

$EA \leftarrow b + \text{EXTZ}(UI \times 4)$

$\text{MEM}(EA, 2) \leftarrow (RS)_{0:15}$

$\text{MEM}(EA+2, 2) \leftarrow (RS)_{32:47}$

D in the instruction mnemonic is $UI \times 4$. The even halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

**Special Registers Altered:**
None

### Code Example

```
if (RA = 0) then b \leftarrow 0
else b \leftarrow (RA)
EA \leftarrow b + (RB)
\text{MEM}(EA, 2) \leftarrow (RS)_{0:15}
\text{MEM}(EA+2, 2) \leftarrow (RS)_{32:47}
```

**Vector Store Word of Two Halfwords from Odd Indexed EVX-form**

```
evstwox RS,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>816</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then $b \leftarrow 0$
else $b \leftarrow (RA)$

$EA \leftarrow b + (RB)$

$\text{MEM}(EA, 2) \leftarrow (RS)_{16:31}$

$\text{MEM}(EA+2, 2) \leftarrow (RS)_{48:63}$

The odd halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

**Special Registers Altered:**
None

### Code Example

```
if (RA = 0) then b \leftarrow 0
else b \leftarrow (RA)
EA \leftarrow b + (RB)
\text{MEM}(EA, 2) \leftarrow (RS)_{16:31}
\text{MEM}(EA+2, 2) \leftarrow (RS)_{48:63}
```

---

**Vector Store Word of Word from Even EVX-form**

```
evstwwe RS,D(RA)
```

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
<th>821</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then $b \leftarrow 0$
else $b \leftarrow (RA)$

$EA \leftarrow b + \text{EXTZ}(UI \times 4)$

$\text{MEM}(EA, 4) \leftarrow (RS)_{0:31}$

D in the instruction mnemonic is $UI \times 4$. The even word of RS is stored in storage addressed by EA.

**Special Registers Altered:**
None

### Code Example

```
if (RA = 0) then b \leftarrow 0
else b \leftarrow (RA)
EA \leftarrow b + (RB)
\text{MEM}(EA, 4) \leftarrow (RS)_{0:31}
```

**Vector Store Word of Word from Even Indexed EVX-form**

```
evstwwex RS,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>824</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

if (RA = 0) then $b \leftarrow 0$
else $b \leftarrow (RA)$

$EA \leftarrow b + (RB)$

$\text{MEM}(EA, 4) \leftarrow (RS)_{0:31}$

The even word of RS is stored in storage addressed by EA.

**Special Registers Altered:**
None

### Code Example

```
if (RA = 0) then b \leftarrow 0
else b \leftarrow (RA)
EA \leftarrow b + (RB)
\text{MEM}(EA, 4) \leftarrow (RS)_{0:31}
```
Vector Store Word of Word from Odd
EVX-form

evstwoo RS,D(RA)

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RS</th>
<th>RA</th>
<th>UI</th>
<th>829</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXT2(UI×4)
MEM(EA, 4) ← (RS)32:63

D in the instruction mnemonic is UI × 4. The odd word of RS is stored in storage addressed by EA.

Special Registers Altered:
None

Vector Store Word of Word from Odd Indexed
EVX-form

evstwwox RS,RA,RB

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>828</th>
</tr>
</thead>
</table>

if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63

The odd word of RS is stored in storage addressed by EA.

Special Registers Altered:
None

Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX-form

evsubfsmiaaw RT,RA

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>///</th>
<th>1227</th>
</tr>
</thead>
</table>

RT0:31 ← (ACC)0:31 - (RA)0:31
RT32:63 ← (ACC)32:63 - (RA)32:63
ACC0:63 ← (RT)0:63

Each word element in RA is subtracted from the corresponding element in the accumulator and the difference is placed into the corresponding RT word and into the accumulator.

Special Registers Altered:
ACC

Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX-form

evsubfssiaaw RT,RA

<table>
<thead>
<tr>
<th>p</th>
<th>4</th>
<th>RT</th>
<th>RA</th>
<th>///</th>
<th>1219</th>
</tr>
</thead>
</table>

temp0:63 ← EXTS((ACC)0:31) - EXTS((RA)0:31)
ovh ← temp31 ⊕ temp32
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)
temp0:63 ← EXTS((ACC)32:63) - EXTS((RA)32:63)
owl ← temp31 ⊕ temp32
RT32:63 ← SATURATE(owl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCRSOVH ← ovh
SPEFSCRSOVOV ← ovl
SPEFSCRSOVS ← SPEFSCRSOVH | ovh
SPEFSCRSOVS ← SPEFSCRSOVOV | ovl

Each signed-integer word element in RA is sign-extended and subtracted from the corresponding sign-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH
**Vector Subtract Unsigned, Modulo, Integer to Accumulator Word   EVX-form**

\[ \text{evsubfumiaaw RT,RA} \]

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>//</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1226</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_{T0:31} \leftarrow (\text{ACC})_{0:31} - (\text{RA})_{0:31} \]
\[ R_{T32:63} \leftarrow (\text{ACC})_{32:63} - (\text{RA})_{32:63} \]
\[ \text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \]

Each unsigned-integer word element in RA is subtracted from the corresponding element in the accumulator and the results are placed in RT and into the accumulator.

Special Registers Altered:
- ACC

**Vector Subtract Unsigned, Saturate, Integer to Accumulator Word   EVX-form**

\[ \text{evsubfusiaaw RT,RA} \]

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>//</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1218</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{temp}_{0:63} \leftarrow \text{EXTZ}(\text{ACC})_{0:31} - \text{EXTZ}(\text{RA})_{0:31} \]
\[ \text{ovh} \leftarrow \text{temp}_{31} \]
\[ R_{T0:31} \leftarrow \text{SATURATE}(\text{ovh}, \text{temp}_{31}, 0x0000_0000, 0x0000_0000, \text{temp}_{32:63}) \]
\[ \text{temp}_{0:63} \leftarrow \text{EXTS}(\text{ACC})_{32:63} - \text{EXTS}(\text{RA})_{32:63} \]
\[ \text{ovl} \leftarrow \text{temp}_{31} \]
\[ R_{T32:63} \leftarrow \text{SATURATE}(\text{ovl}, \text{temp}_{31}, 0x0000_0000, 0x0000_0000, \text{temp}_{32:63}) \]
\[ \text{ACC}_{0:63} \leftarrow (\text{RT})_{0:63} \]

\[ \text{SPEFSCROVh} \leftarrow \text{ovh} \]
\[ \text{SPEFSCROV} \leftarrow \text{ovl} \]
\[ \text{SPEFSCROVh} \leftarrow \text{SPEFSCROV} | \text{ovh} \]
\[ \text{SPEFSCROV} \leftarrow \text{SPEFSCROV} | \text{ovl} \]

Each unsigned-integer word element in RA is zero-extended and subtracted from the corresponding zero-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
- ACC OV OVH SOV SOVH

**Vector Subtract from Word   EVX-form**

\[ \text{evsubf bw RT,RA,RB} \]

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1218</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_{T0:31} \leftarrow (\text{RB})_{0:31} - (\text{RA})_{0:31} \]
\[ R_{T32:63} \leftarrow (\text{RB})_{32:63} - (\text{RA})_{32:63} \]

Each signed-integer element of RA is subtracted from the corresponding element of RB and the results are placed in RT.

Special Registers Altered:
- None

**Vector Subtract Immediate from Word   EVX-form**

\[ \text{evsubf bw RT,UI,RB} \]

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>UI</th>
<th>RB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1218</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_{T0:31} \leftarrow (\text{RB})_{0:31} - \text{EXTZ}(\text{UI}) \]
\[ R_{T32:63} \leftarrow (\text{RB})_{32:63} - \text{EXTZ}(\text{UI}) \]

UI is zero-extended and subtracted from both the high and low elements of RB. Note that the same value is subtracted from both elements of the register.

Special Registers Altered:
- None

**Vector XOR   EVX-form**

\[ \text{evxor RT,RA,RB} \]

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1218</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_{T0:31} \leftarrow (\text{RA})_{0:31} \oplus (\text{RB})_{0:31} \]
\[ R_{T32:63} \leftarrow (\text{RA})_{32:63} \oplus (\text{RB})_{32:63} \]

Each element of RA and RB is exclusive-ORed. The results are placed in RT.

Special Registers Altered:
- None
Chapter 9. Embedded Floating-Point
[Category: SPE.Embedded Float Scalar Double]
[Category: SPE.Embedded Float Scalar Single]
[Category: SPE.Embedded Float Vector]

9.1 Overview

The Embedded Floating-Point categories require the implementation of the Signal Processing Engine (SPE) category and consist of three distinct categories:

- Embedded vector single-precision floating-point (SPE.Embedded Float Vector [SP.FV])
- Embedded scalar single-precision floating-point (SPE.Embedded Float Scalar Single [SP.FS])
- Embedded scalar double-precision floating-point (SPE.Embedded Float Scalar Double [SP.FD])

Although each of these may be implemented independently, they are defined in a single chapter because it is likely that they may be implemented together.

References to Embedded Floating-Point categories, Embedded Floating-Point instructions, or Embedded Floating-Point operations apply to all 3 categories.

Single-precision floating-point is handled by the SPE.Embedded Float Vector and SPE.Embedded Float Scalar Single categories; double-precision floating-point is handled by the SPE.Embedded Float Scalar Double category.
9.2 Programming Model

Embedded floating-point operations are performed in the GPRs of the processor.

The SPE.Embedded Float Vector and SPE.Embedded Float Scalar Double categories require a GPR register file with thirty-two 64-bit registers as required by the Signal Processing Engine category.

The SPE.Embedded Float Scalar Single category requires a GPR register file with thirty-two 32-bit registers. When implemented with a 64-bit register file on a 32-bit implementation, instructions in this category only use and modify bits 32:63 of the GPR. In this case, bits 0:31 of the GPR are left unchanged by the operation. For 64-bit implementations, bits 0:31 are unchanged after the operation.

Instructions in the SPE.Embedded Float Scalar Double category operate on the entire 64 bits of the GPRs.

Instructions in the SPE.Embedded Float Vector category operate on the entire 64 bits of the GPRs as well, but contain two 32-bit data items that are operated on independently of each other in a SIMD fashion. The format of both data items is the same as the format of a data item in the SPE.Embedded Float Scalar Single category. The data item contained in bits 0:31 is called the 'high word'. The data item contained in bits 32:63 is called the 'low word'.

There are no record forms of Embedded Floating-Point instructions. Embedded Floating-Point Compare instructions treat NaNs, Infinity, and Denorm as normalized numbers for the comparison calculation when default results are provided.

9.2.1 Signal Processing Embedded Floating-Point Status and Control Register (SPEFSCR)

Status and control for the Embedded Floating-Point categories uses the SPEFSCR. This register is defined by the Signal Processing Engine category in Section 8.3.4. Status and control bits are shared for Embedded Floating-Point and SPE operations. Instructions in the SPE.Embedded Float Vector category affect both the high element (bits 34:39) and low element floating-point status flags (bits 50:55). Instructions in the SPE.Embedded Float Scalar Double and SPE.Embedded Float Scalar Single categories affect only the low element floating-point status flags and leave the high element floating-point status flags undefined.

9.2.2 Floating-Point Data Formats

Single-precision floating-point data elements are 32 bits wide with 1 sign bit (s), 8 bits of biased exponent (e) and 23 bits of fraction (f). Double-precision floating-point data elements are 64 bits wide with 1 sign bit (s), 11 bits of biased exponent (e) and 52 bits of fraction (f).

In the IEEE 754 specification, floating-point values are represented in a format consisting of three explicit fields (sign field, biased exponent field, and fraction field) and an implicit hidden bit.

![Figure 130. Floating-Point Data Format](image)

For single-precision normalized numbers, the biased exponent value e lies in the range of 1 to 254 corresponding to an actual exponent value E in the range -126 to +127. For double-precision normalized numbers, the biased exponent value e lies in the range of 1 to 2046 corresponding to an actual exponent value E in the range -1022 to +1023. With the hidden bit implied to be '1' (for normalized numbers), the value of the number is interpreted as follows:

\[ (-1)^s \times 2^e \times (1.fraction) \]

where E is the unbiased exponent and 1.fraction is the mantissa (or significand) consisting of a leading '1' (the hidden bit) and a fractional part (fraction field). For the single-precision format, the maximum positive normalized number (pmax) is represented by the encoding 0x7F7FFFFF which is approximately 3.4E+38 (2^{128}), and the minimum positive normalized value (pmin) is represented by the encoding 0x00800000 which is approximately 1.2E-38 (2^{-126}). For the double-precision format, the maximum positive normalized number (pmax) is represented by the encoding 0x7fecccccccccccccccc which is approximately 7.1E+307 (2^{1024}), and the minimum positive normalized value (pmin) is represented by the encoding 0x0010000000000000 which is approximately 2.2E-308 (2^{-1022}).

Two specific values of the biased exponent are reserved (0 and 255 for single-precision; 0 and 2047 for double-precision) for encoding special values of +0, -0, +infinity, -infinity, and NaNs.

Zeros of both positive and negative sign are represented by a biased exponent value e of 0 and a fraction f which is 0.

Infinites of both positive and negative sign are represented by a maximum exponent field value (255 for single-precision, 2047 for double-precision) and a fraction which is 0.
Denormalized numbers of both positive and negative sign are represented by a biased exponent value \( e \) of 0 and a fraction \( f \), which is nonzero. For these numbers, the hidden bit is defined by the IEEE 754 standard to be 0. This number type is not directly supported in hardware. Instead, either a software interrupt handler is invoked, or a default value is defined.

Not-a-Numbers (NaNs) are represented by a maximum exponent field value (255 for single-precision, 2047 for double-precision) and a fraction \( f \), which is nonzero. For these numbers, the hidden bit is defined by the IEEE 754 standard to be 0. This number type is not directly supported in hardware. Instead, either a software interrupt handler is invoked, or a default value is defined.

9.2.3 Exception Conditions

9.2.3.1 Denormalized Values on Input
Any denormalized value used as an operand may be truncated by the implementation to a properly signed zero value.

9.2.3.2 Embedded Floating-Point Overflow and Underflow
Defining \( p_{max} \) to be the most positive normalized value (farthest from zero), \( p_{min} \) the smallest positive normalized value (farthest from zero) and \( n_{max} \) the smallest normalized negative value (closest to zero), an overflow is said to have occurred if the numerically correct result \( r \) of an instruction is such that \( r > p_{max} \) or \( r < n_{max} \). An underflow is said to have occurred if the numerically correct result of an instruction is such that \( 0 < r < p_{min} \) or \( n_{min} < r < 0 \). In this case, \( r \) may be denormalized, or may be smaller than the smallest denormalized number.

The Embedded Floating-Point categories do not produce +Infinity, -Infinity, NaN, or denormalized numbers. If the result of an instruction overflows and Embedded Floating-Point Overflow exceptions are disabled (SPEFSCR\_FVFE=0), \( p_{max} \) or \( n_{max} \) is generated as the result of that instruction depending upon the sign of the result. If the result of an instruction underflows and Embedded Floating-Point Underflow exceptions are disabled (SPEFSCR\_FUNE=0), +0 or -0 is generated as the result of that instruction based upon the sign of the result.

If an overflow occurs, SPEFSCR\_FOVF FOVFH are set as appropriate, or if an underflow occurs, SPEFSCR\_FUNF FUNFH are set as appropriate. If either Embedded Floating-Point Underflow or Embedded Floating-Point Overflow exceptions are enabled and a corresponding status bit is 1, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

9.2.3.3 Embedded Floating-Point Invalid Operation/Input Errors
Embedded Floating-Point Invalid Operation/Input errors occur when an operand to an operation contains an invalid input value. If any of the input values are Infinity, Denorm, or NaN, or for an Embedded Floating-Point Divide instruction both operands are +/-0, SPEFSCR\_F. INV\_FINVH are set to 1 appropriately, and SPEFSCR\_FXH FG FXH are set to 0 appropriately. If SPEFSCR\_F, INVE=1, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

9.2.3.4 Embedded Floating-Point Round (Inexact)
If any result element of an Embedded Floating-Point instruction is inexact, or overflows but Embedded Floating-Point Overflow exceptions are disabled, or underflows but Embedded Floating-Point Underflow exceptions are disabled, and no higher priority interrupt occurs, SPEFSCR\_FINXS is set to 1. If the Embedded Floating-Point Round (inexact) exception is enabled, an Embedded Floating-Point Round interrupt occurs. In this case, the destination register is updated with the truncated result(s). The SPEFSCR\_FGH FXH FG FX bits are properly updated to allow rounding to be performed in the interrupt handler.

SPEFSCR\_FG FX (SPEFSCR\_FGH FXH) are set to 0 if an Embedded Floating-Point Data interrupt is taken due to overflow, underflow, or if an Embedded Floating-Point Invalid Operation/Input error is signaled for the low (high) element (regardless of SPEFSCR\_FINVE).

9.2.3.5 Embedded Floating-Point Divide by Zero
If an Embedded Floating-Point Divide instruction executes and an Embedded Floating-Point Invalid Operation/Input error does not occur and the instruction is executed with a +/-0 divisor value and a finite normalized nonzero dividend value, an Embedded Floating-Point Divide By Zero exception occurs and SPEFSCR\_FDBZ FDBZH are set appropriately. If Embedded Floating-Point Divide By Zero exceptions are enabled, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

Programming Note
On some implementations, operations that result in overflow or underflow are likely to take significantly longer than operations that do not. For example, these operations may cause a system error handler to be invoked; on such implementations, the system error handler updates the overflow bits appropriately.
interrupt is then taken and the destination register is not updated.

9.2.3.6 Default Results

Default results are generated when an Embedded Floating-Point Invalid Operation/Input Error, Embedded Floating-Point Overflow, Embedded Floating-Point Underflow, or Embedded Floating-Point Divide by Zero occurs on an Embedded Floating-Point operation. Default results provide a normalized value as a result of the operation. In general, Denorm results and underflows are set to 0 and overflows are saturated to the maximum representable number.

Default results produced for each operation are described in Section 9.4, “Embedded Floating-Point Results Summary”.

9.2.4 IEEE 754 Compliance

The Embedded Floating-Point categories require a floating-point system as defined in the ANSI/IEEE Standard 754-1985 but may rely on software support in order to conform fully with the standard. Thus, whenever an input operand of the Embedded Floating-Point instruction has data values that are +Infinity, -Infinity, Denormalized, NaN, or when the result of an operation produces an overflow or an underflow, an Embedded Floating-Point Data interrupt may be taken and the interrupt handler is responsible for delivering IEEE 754 compliant behavior if desired.

When Embedded Floating-Point Invalid Operation/Input Error exceptions are disabled (SPEFSCRFINVE = 0), default results are produced by the hardware when an Infinity, Denormalized, or NaN input is received, or for the operation 0/0. When Embedded Floating-Point Underflow exceptions are disabled (SPEFSCRFUNFE = 0) and the result of a floating-point operation underflows, a signed zero result is produced. The Embedded Floating-Point Round (Inexact) exception is also signaled for this condition. When Embedded Floating-Point Overflow exceptions are disabled (SPEFSCRFOVFE = 0) and the result of a floating-point operation overflows, a pmax or nmax result is produced. The Embedded Floating-Point Round (Inexact) exception is also signaled for this condition. An exception enable flag (SPEFSCRFINXE) is also provided for generating an Embedded Floating-Point Round interrupt when an inexact result is produced, to allow a software handler to conform to the IEEE 754 standard. An Embedded Floating-Point Divide By Zero exception enable flag (SPEFSCRFDBZE) is provided for generating an Embedded Floating-Point Data interrupt when a divide by zero operation is attempted to allow a software handler to conform to the IEEE 754 standard. All of these exceptions may be disabled, and the hardware will then deliver an appropriate default result.

The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. If both operands have the same sign, the sign of the result is the same as the sign of the operands. This includes subtraction which is addition with the negation of the sign of the second operand. The sign of the result of an addition operation with operands of differing signs for which the result is zero is positive except when rounding to negative infinity. Thus -0 + -0 = -0, and all other cases which result in a zero value give +0 unless the rounding mode is round to negative infinity.

--- Programming Note ---

Note that when exceptions are disabled and default results computed, operations having input values that are denormalized may provide different results on different implementations. An implementation may choose to use the denormalized value or a zero value for any computation. Thus a computational operation involving a denormalized value and a normal value may return different results depending on the implementation.

9.2.4.1 Sticky Bit Handling For Exception Conditions

The SPEFSCR register defines sticky bits for retaining information about exception conditions that are detected. There are 5 sticky bits (FINXS, FINVS, FDBZS, FUNFS and FOVFS) that can be used to help provide IEEE 754 compliance. The sticky bits represent the combined ‘or’ of all the previous status bits produced from any Embedded Floating-Point operation since the last time software zeroed the sticky bit. The hardware will never set a sticky bit to 0.
9.3 Embedded Floating-Point Instructions

9.3.1 Load/Store Instructions

*Embedded Floating-Point* instructions use GPRs to hold and operate on floating-point values. The *Embedded Floating-Point* categories do not define *Load* and *Store* instructions to move the data to and from memory, but instead rely on existing instructions in Book I to load and store data.

### Vector Floating-Point Single-Precision

#### Absolute Value

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>evfsabs</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccc}
4 & 9 & 11 & 16 & 21 & 31 & 644 & 0 \\
\end{array}
\]

\[
RT_{0..31} \leftarrow 0b0 || (RA)_{1..31} \\
RT_{32..63} \leftarrow 0b0 || (RA)_{33..63}
\]

The sign bit of each element in register RA is set to 0 and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

#### Negative Absolute Value

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>evfsnabs</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccc}
4 & 9 & 11 & 16 & 21 & 31 & 645 & 0 \\
\end{array}
\]

\[
RT_{0..31} \leftarrow 0b1 || (RA)_{1..31} \\
RT_{32..63} \leftarrow 0b1 || (RA)_{33..63}
\]

The sign bit of each element in register RA is set to 1 and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

#### Negate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>evfsneg</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccc}
0 & 9 & 11 & 16 & 21 & 31 & 646 & 0 \\
\end{array}
\]

\[
RT_{0..31} \leftarrow \neg (RA)_{0} || (RA)_{1..31} \\
RT_{32..63} \leftarrow \neg (RA)_{32} || (RA)_{33..63}
\]

The sign bit of each element in register RA is complemented and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

9.3.2 SPE.Embedded Float Vector Instructions [Category: SPE.Embedded Float Vector]

All SPE.Embedded Float Vector instructions are single-precision. There are no vector floating-point double-precision instructions.
### Vector Floating-Point Single-Precision Add

**EVX-form**

```
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>640</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>
```

```
RT_{0:31} ← (RA)_{0:31} + (RB)_{0:31}
RT_{32:63} ← (RA)_{32:63} + (RB)_{32:63}
```

Each single-precision floating-point element of register RA is added to the corresponding element of register RB and the results are stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of register RT.

**Special Registers Altered:**

- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- FINXS
- FOVF
- FOVFH
- FOVFS
- FUNF
- FUNFH
- FUNFS

### Vector Floating-Point Single-Precision Subtract

**EVX-form**

```
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>641</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>
```

```
RT_{0:31} ← (RA)_{0:31} - (RB)_{0:31}
RT_{32:63} ← (RA)_{32:63} - (RB)_{32:63}
```

Each single-precision floating-point element of register RB is subtracted from the corresponding element of register RA and the results are stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of register RT.

**Special Registers Altered:**

- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- FINXS
- FOVF
- FOVFH
- FOVFS
- FUNF
- FUNFH
- FUNFS

### Vector Floating-Point Single-Precision Multiply

**EVX-form**

```
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>648</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>
```

```
RT_{0:31} ← (RA)_{0:31} × (RB)_{0:31}
RT_{32:63} ← (RA)_{32:63} × (RB)_{32:63}
```

Each single-precision floating-point element of register RA is multiplied with the corresponding element of register RB and the result is stored in register RT.

**Special Registers Altered:**

- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- FINXS
- FOVF
- FOVFH
- FOVFS
- FUNF
- FUNFH
- FUNFS

### Vector Floating-Point Single-Precision Divide

**EVX-form**

```
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>649</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>
```

```
RT_{0:31} ← (RA)_{0:31} ÷ (RB)_{0:31}
RT_{32:63} ← (RA)_{32:63} ÷ (RB)_{32:63}
```

Each single-precision floating-point element of register RA is divided by the corresponding element of register RB and the result is stored in register RT.

**Special Registers Altered:**

- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- FINXS
- FDBZ
- FDBZH
- FDBZS
- FOVF
- FOVFH
- FOVFS
- FUNF
- FUNFH
- FUNFS
**Vector Floating-Point Single-Precision**  
**Compare Greater Than**  
**EVX-form**

```
evfscmpgt BF ,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>652</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p</td>
<td>t1</td>
<td>t0</td>
<td></td>
</tr>
</tbody>
</table>

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If RA0:31 is greater than RB0:31, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If RA32:63 is greater than RB32:63, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**
- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- CR field BF

---

**Vector Floating-Point Single-Precision**  
**Compare Less Than**  
**EVX-form**

```
evfscmplt BF ,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>653</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p</td>
<td>t1</td>
<td>t0</td>
<td></td>
</tr>
</tbody>
</table>

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If RA0:31 is less than RB0:31, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If RA32:63 is less than RB32:63, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**
- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- CR field BF
Vector Floating-Point Single-Precision

Compare Equal EVX-form

```
evfscmpeq BF,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF //</th>
<th>RA</th>
<th>RB</th>
<th>654</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>31</td>
</tr>
</tbody>
</table>

ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If RA0:31 is equal to RB0:31, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If RA32:63 is equal to RB32:63, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of 'e' and 'f' directly.

Special Registers Altered:

- FINV FINVH FINVS
- FGH FXH FG FX
- CR field BF

Vector Floating-Point Single-Precision

Test Greater Than EVX-form

```
evfststgt BF,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF //</th>
<th>RA</th>
<th>RB</th>
<th>668</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>31</td>
</tr>
</tbody>
</table>

ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If RA0:31 is greater than RB0:31, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If RA32:63 is greater than RB32:63, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of 'e' and 'f' directly.

No exceptions are taken during the execution of evfststgt.

Special Registers Altered:

- CR field BF

Programming Note

In an implementation, the execution of evfststgt is likely to be faster than the execution of evfscmpgt; however, if strict IEEE 754 compliance is required, the program should use evfscmpgt.
Vector Floating-Point Single-Precision
Test Less Than
EVX-form

`evfststlt BF,RA,RB`

```
<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>669</th>
</tr>
</thead>
</table>
```

ah ← (RA)_{0:31}
al ← (RA)_{32:63}
bh ← (RB)_{0:31}
bl ← (RB)_{32:63}
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR_{4×BF:4×BF+3} ← ch || cl || (ch | cl) || (ch & cl)

Each element of register RA is compared with the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If RA_{0:31} is less than RB_{0:31}, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If RA_{32:63} is less than RB_{32:63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are taken during the execution of `evfststlt`.

Special Registers Altered:
CR field BF

**Programming Note**
In an implementation, the execution of `evfststlt` is likely to be faster than the execution of `evfscmplt`; however, if strict IEEE 754 compliance is required, the program should use `evfscmplt`.

Vector Floating-Point Single-Precision
Test Equal
EVX-form

`evfststeq BF,RA,RB`

```
<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>670</th>
</tr>
</thead>
</table>
```

ah ← (RA)_{0:31}
al ← (RA)_{32:63}
bh ← (RB)_{0:31}
bl ← (RB)_{32:63}
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR_{4×BF:4×BF+3} ← ch || cl || (ch | cl) || (ch & cl)

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If RA_{0:31} is equal to RB_{0:31}, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If RA_{32:63} is equal to RB_{32:63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are taken during the execution of `evfststeq`.

**Special Registers Altered:**
CR field BF

**Programming Note**
In an implementation, the execution of `evfststeq` is likely to be faster than the execution of `evfscmpeq`; however, if strict IEEE 754 compliance is required, the program should use `evfscmpeq`.
Vector Convert Floating-Point
Single-Precision from Signed Integer

**EVX-form**

\[ \text{evfscfsi} \quad \text{RT}, \text{RB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>//</th>
<th>RB</th>
<th>657</th>
</tr>
</thead>
</table>

- \( \text{RT}_{0:31} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{0:31}, \text{S}, \text{HI}, \text{I}) \)
- \( \text{RT}_{32:63} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{32:63}, \text{S}, \text{LO}, \text{I}) \)

Each signed integer element of register RB is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding element of register RT.

**Special Registers Altered:**
- FG, FH, FG, FX, FINXS

Vector Convert Floating-Point
Single-Precision from Unsigned Integer

**EVX-form**

\[ \text{evfscfui} \quad \text{RT}, \text{RB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>//</th>
<th>RB</th>
<th>656</th>
</tr>
</thead>
</table>

- \( \text{RT}_{0:31} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{0:31}, \text{U}, \text{HI}, \text{I}) \)
- \( \text{RT}_{32:63} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{32:63}, \text{U}, \text{LO}, \text{I}) \)

Each unsigned integer element of register RB is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

**Special Registers Altered:**
- FG, FH, FG, FX, FINXS

Vector Convert Floating-Point
Single-Precision from Signed Fraction

**EVX-form**

\[ \text{evfscfsf} \quad \text{RT}, \text{RB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>//</th>
<th>RB</th>
<th>659</th>
</tr>
</thead>
</table>

- \( \text{RT}_{0:31} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{0:31}, \text{S}, \text{HI}, \text{F}) \)
- \( \text{RT}_{32:63} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{32:63}, \text{S}, \text{LO}, \text{F}) \)

Each signed fractional element of register RB is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

**Special Registers Altered:**
- FG, FH, FG, FX, FINXS

Vector Convert Floating-Point
Single-Precision from Unsigned Fraction

**EVX-form**

\[ \text{evfscfuf} \quad \text{RT}, \text{RB} \]

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>//</th>
<th>RB</th>
<th>658</th>
</tr>
</thead>
</table>

- \( \text{RT}_{0:31} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{0:31}, \text{U}, \text{HI}, \text{F}) \)
- \( \text{RT}_{32:63} \rightarrow \text{CnvtI32ToF32}((\text{RB})_{32:63}, \text{U}, \text{LO}, \text{F}) \)

Each unsigned fractional element of register RB is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

**Special Registers Altered:**
- FG, FH, FG, FX, FINXS
**Vector Convert Floating-Point Single-Precision to Signed Integer**

**EVX-form**

\[
\text{evfsctsi \ RT,RB}
\]

```
RT_{0:31} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{0:31}, S, HI, RND, I)
RT_{32:63} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{32:63}, S, LO, RND, I)
```

Each single-precision floating-point element in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**

FINV FINVH FINVS
FGH FXH FG FX FINXS

---

**Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero**

**EVX-form**

\[
\text{evfsctsiz \ RT,RB}
\]

```
RT_{0:31} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{0:31}, S, HI, ZER, I)
RT_{32:63} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{32:63}, S, LO, ZER, I)
```

Each single-precision floating-point element in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**

FINV FINVH FINVS
FGH FXH FG FX FINXS

---

**Vector Convert Floating-Point Single-Precision to Unsigned Integer**

**EVX-form**

\[
\text{evfsctui \ RT,RB}
\]

```
RT_{0:31} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{0:31}, U, HI, RND, I)
RT_{32:63} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{32:63}, U, LO, RND, I)
```

Each single-precision floating-point element in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**

FINV FINVH FINVS
FGH FXH FG FX FINXS

---

**Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero**

**EVX-form**

\[
\text{evfsctuiz \ RT,RB}
\]

```
RT_{0:31} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{0:31}, U, HI, ZER, I)
RT_{32:63} \leftarrow \text{CnvtfP32ToI32Sat}((RB)_{32:63}, U, LO, ZER, I)
```

Each single-precision floating-point element in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**

FINV FINVH FINVS
FGH FXH FG FX FINXS
Vector Convert Floating-Point  
Single-Precision to Signed Fraction  
**EVX-form**

```
evfsctsf RT,RB

| 4 | RT | /// | RB | 663 |
```

\[
RT_{0:31} \leftarrow \text{CnvtFP32ToI32Sat}((RB)_{0:31}, S, \text{HI}, \text{RND}, F)
\]

Each single-precision floating-point element in register \( RB \) is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit signed fraction. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- FINXS

Vector Convert Floating-Point  
Single-Precision to Unsigned Fraction  
**EVX-form**

```
evfsctuf RT,RB

| 4 | RT | /// | RB | 662 |
```

\[
RT_{0:31} \leftarrow \text{CnvtFP32ToI32Sat}((RB)_{0:31}, U, \text{HI}, \text{RND}, F)
\]

Each single-precision floating-point element in register \( RB \) is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVH
- FINVS
- FGH
- FXH
- FG
- FX
- FINXS
9.3.3 SPE.Embedded Float Scalar Single Instructions

[Category: SPE.Embedded Float Scalar Single]

**Floating-Point Single-Precision Absolute Value EVX-form**

<table>
<thead>
<tr>
<th>efsabs</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>///</td>
<td>708</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow 0b0 || (RA)_{33:63} \]

The sign bit of the low element of register RA is set to 0 and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

**Special Registers Altered:**

None

**Floating-Point Single-Precision Negative Absolute Value EVX-form**

<table>
<thead>
<tr>
<th>efsnabs</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>///</td>
<td>709</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow 0b1 || (RA)_{33:63} \]

The sign bit of the low element of register RA is set to 1 and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

**Special Registers Altered:**

None

**Floating-Point Single-Precision Negate EVX-form**

<table>
<thead>
<tr>
<th>efsneg</th>
<th>RT,RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RT</td>
</tr>
<tr>
<td>11</td>
<td>RA</td>
</tr>
<tr>
<td>///</td>
<td>710</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \neg (RA)_{32} || (RA)_{33:63} \]

The sign bit of the low element of register RA is complemented and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

**Special Registers Altered:**

None
### Floating-Point Single-Precision Add

**EVX-form**

<table>
<thead>
<tr>
<th>efsadd</th>
<th>RT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>p</td>
<td>f1</td>
</tr>
<tr>
<td>32:63</td>
<td>704</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} + sp (RB)_{32:63} \]

The low element of register RA is added to the low element of register RB and the result is stored in the low element of register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

**Special Registers Altered:**
- FINV
- FINVS
- FOVF
- FOVFS
- FUNF
- FUNFS
- FG
- FX
- FINXS

### Floating-Point Single-Precision Subtract

**EVX-form**

<table>
<thead>
<tr>
<th>efssub</th>
<th>RT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>p</td>
<td>f1</td>
</tr>
<tr>
<td>32:63</td>
<td>705</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} - sp (RB)_{32:63} \]

The low element of register RB is subtracted from the low element of register RA and the result is stored in the low element of register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

**Special Registers Altered:**
- FINV
- FINVS
- FOVF
- FOVFS
- FUNF
- FUNFS
- FG
- FX
- FINXS

### Floating-Point Single-Precision Multiply

**EVX-form**

<table>
<thead>
<tr>
<th>efsmul</th>
<th>RT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>p</td>
<td>f1</td>
</tr>
<tr>
<td>32:63</td>
<td>712</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} \times sp (RB)_{32:63} \]

The low element of register RA is multiplied by the low element of register RB and the result is stored in the low element of register RT.

**Special Registers Altered:**
- FINV
- FINVS
- FOVF
- FOVFS
- FUNF
- FUNFS
- FG
- FX
- FINXS

### Floating-Point Single-Precision Divide

**EVX-form**

<table>
<thead>
<tr>
<th>efsdiv</th>
<th>RT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>p</td>
<td>f1</td>
</tr>
<tr>
<td>32:63</td>
<td>713</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow (RA)_{32:63} ÷ sp (RB)_{32:63} \]

The low element of register RA is divided by the low element of register RB and the result is stored in the low element of register RT.

**Special Registers Altered:**
- FINV
- FINVS
- FG
- FX
- FINXS
- FDBZ
- FDBZS
- FOVF
- FOVFS
- FUNF
- FUNFS
### Floating-Point Single-Precision Compare

#### Greater Than

**EVX-form**

```
efscmpgt BF,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>716</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{al} &\leftarrow (\text{RA})_{32:63} \\
\text{bl} &\leftarrow (\text{RB})_{32:63} \\
\text{if} \ (\text{al} > \text{bl}) \ &\text{then} \ cl &\leftarrow 1 \\
\text{else} \ cl &\leftarrow 0 \\
\text{CR}_{4\times\text{BF}+4} &\leftarrow \text{undefined} || cl || \text{undefined} || \text{undefined} \\
\end{align*}
\]

The low element of register RA is compared against the low element of register RB. The results of the comparisons are placed into CR field BF. If RA\text{32-63} is greater than RB\text{32-63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**

- FINV
- FINVS
- FG
- FX
- CR field BF

#### Less Than

**EVX-form**

```
efscmplt BF,RA,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>717</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{al} &\leftarrow (\text{RA})_{32:63} \\
\text{bl} &\leftarrow (\text{RB})_{32:63} \\
\text{if} \ (\text{al} < \text{bl}) \ &\text{then} \ cl &\leftarrow 1 \\
\text{else} \ cl &\leftarrow 0 \\
\text{CR}_{4\times\text{BF}+4} &\leftarrow \text{undefined} || cl || \text{undefined} || \text{undefined} \\
\end{align*}
\]

The low element of register RA is compared against the low element of register RB. If RA\text{32-63} is less than RB\text{32-63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**

- FINV
- FINVS
- FG
- FX
- CR field BF
Floating-Point Single-Precision Compare Equal

`efscmpeq BF,RA,RB`

```
4
BF // RA  RB  718
0  32:63 35
1  32:63 35
```

```
al ← (RA)_{32:63}
bl ← (RB)_{32:63}
if (al = bl) then cl ← 1
else cl ← 0
CR_{4×BF:4×BF+3} ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If RA_{32:63} is equal to RB_{32:63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

Special Registers Altered:

| FINV | FINVS |
| FG  | FX   |
| CR field BF |

Floating-Point Single-Precision Test Greater Than

`efststgt BF,RA,RB`

```
4
BF // RA  RB  732
0  32:63 35
1  32:63 35
```

```
al ← (RA)_{32:63}
bl ← (RB)_{32:63}
if (al > bl) then cl ← 1
else cl ← 0
CR_{4×BF:4×BF+3} ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If RA_{32:63} is greater than RB_{32:63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are generated during the execution of `efststgt`.

Special Registers Altered:

| CR field BF |

Programming Note

In an implementation, the execution of `efststgt` is likely to be faster than the execution of `efscmpgt`; however, if strict IEEE 754 compliance is required, the program should use `efscmpgt`. 
Floating-Point Single-Precision Test Less Than EVX-form

\texttt{efstslt BF,RA,RB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & BF & // & RA & RB & 733 \\
\hline
\end{tabular}

\begin{itemize}
\item \texttt{al \leftarrow (RA)_{32:63}}
\item \texttt{bl \leftarrow (RB)_{32:63}}
\item If \texttt{al < bl} then \texttt{cl \leftarrow 1}
\item Else \texttt{cl \leftarrow 0}
\item \texttt{CR4xFB:4xFB+3 \leftarrow undefined || cl || undefined || undefined}
\end{itemize}

The low element of register RA is compared against the low element of register RB. If RA_{32:63} is less than RB_{32:63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are generated during the execution of \texttt{efstslt}.

Special Registers Altered:

- CR field BF

Programming Note

In an implementation, the execution of \texttt{efstslt} is likely to be faster than the execution of \texttt{efscmplt}; however, if strict IEEE 754 compliance is required, the program should use \texttt{efscmplt}.

Floating-Point Single-Precision Test Equal EVX-form

\texttt{efststeq BF,RA,RB}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
4 & BF & // & RA & RB & 734 \\
\hline
\end{tabular}

\begin{itemize}
\item \texttt{al \leftarrow (RA)_{32:63}}
\item \texttt{bl \leftarrow (RB)_{32:63}}
\item If \texttt{al = bl} then \texttt{cl \leftarrow 1}
\item Else \texttt{cl \leftarrow 0}
\item \texttt{CR4xFB:4xFB+3 \leftarrow undefined || cl || undefined || undefined}
\end{itemize}

The low element of register RA is compared against the low element of register RB. If RA_{32:63} is equal to RB_{32:63}, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are generated during the execution of \texttt{efststeq}.

Special Registers Altered:

- CR field BF

Programming Note

In an implementation, the execution of \texttt{efststeq} is likely to be faster than the execution of \texttt{efscmpeq}; however, if strict IEEE 754 compliance is required, the program should use \texttt{efscmpeq}.
### Convert Floating-Point Single-Precision from Signed Integer EVX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efscfsi</td>
<td>4 RT /// 721</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtI32ToFp32}((\text{RB})_{32:63}, S, LO, I) \]

The signed integer low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

**Special Registers Altered:**
- FINXS
- FG
- FX

### Convert Floating-Point Single-Precision from Unsigned Integer EVX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efscfui</td>
<td>4 RT /// 720</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtI32ToFp32}((\text{RB})_{32:63}, U, LO, I) \]

The unsigned integer low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

**Special Registers Altered:**
- FINXS
- FG
- FX

### Convert Floating-Point Single-Precision from Signed Fraction EVX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efscfsf</td>
<td>4 RT /// 723</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtI32ToFp32}((\text{RB})_{32:63}, S, LO, F) \]

The signed fractional low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

**Special Registers Altered:**
- FINXS
- FG
- FX

### Convert Floating-Point Single-Precision from Unsigned Fraction EVX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efscfuf</td>
<td>4 RT /// 722</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtI32ToFp32}((\text{RB})_{32:63}, U, LO, F) \]

The unsigned fractional low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

**Special Registers Altered:**
- FINXS
- FG
- FX

### Convert Floating-Point Single-Precision to Signed Integer EVX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efscitsi</td>
<td>4 RT /// 725</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtFp32ToI32Sat}((\text{RB})_{32:63}, S, LO, RND, I) \]

The single-precision floating-point low element in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

### Convert Floating-Point Single-Precision to Unsigned Integer EVX-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efscitui</td>
<td>4 RT /// 724</td>
</tr>
</tbody>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtFp32ToI32Sat}((\text{RB})_{32:63}, U, LO, RND, I) \]

The single-precision floating-point low element in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX
Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero

**EVX-form**

\[ \text{efsctsiz} \quad RT, RB \]

\[ \begin{array}{cccccc}
4 & R & 11 & 16 & 21 & 31 \\
\hline
\end{array} \]

\[ RT_{32:63} \leftarrow \text{CnvtFP32ToI32Sat}((RB)_{32:63}, S, LO, ZER, I) \]

The single-precision floating-point low element in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero

**EVX-form**

\[ \text{efsctuz} \quad RT, RB \]

\[ \begin{array}{cccccc}
4 & R & 11 & 16 & 21 & 31 \\
\hline
\end{array} \]

\[ RT_{32:63} \leftarrow \text{CnvtFP32ToI32Sat}((RB)_{32:63}, U, LO, ZER, I) \]

The single-precision floating-point low element in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

Convert Floating-Point Single-Precision to Signed Fraction

**EVX-form**

\[ \text{efsctsf} \quad RT, RB \]

\[ \begin{array}{cccccc}
4 & R & 11 & 16 & 21 & 31 \\
\hline
\end{array} \]

\[ RT_{32:63} \leftarrow \text{CnvtFP32ToI32Sat}((RB)_{32:63}, S, LO, RND, F) \]

The single-precision floating-point low element in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

Convert Floating-Point Single-Precision to Unsigned Fraction

**EVX-form**

\[ \text{efsctuf} \quad RT, RB \]

\[ \begin{array}{cccccc}
4 & R & 11 & 16 & 21 & 31 \\
\hline
\end{array} \]

\[ RT_{32:63} \leftarrow \text{CnvtFP32ToI32Sat}((RB)_{32:63}, U, LO, RND, F) \]

The single-precision floating-point low element in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX
9.3.4 SPE.Embedded Float Scalar Double Instructions
[Category: SPE.Embedded Float Scalar Double]

Floating-Point Double-Precision Absolute Value

\[ \text{efdabs } \text{RT}, \text{RA} \]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
| & 4 & \text{RT} & \text{RA} & /// & 740 \\
\hline
\end{array}
\]

\[ \text{RT}_{0:63} \leftarrow 0b0 || \langle \text{RA} \rangle_{1:63} \]

The sign bit of register RA is set to 0 and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Double-Precision Negative Absolute Value

\[ \text{efdnabs } \text{RT}, \text{RA} \]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
| & 4 & \text{RT} & \text{RA} & /// & 741 \\
\hline
\end{array}
\]

\[ \text{RT}_{0:63} \leftarrow 0b1 || \langle \text{RA} \rangle_{1:63} \]

The sign bit of register RA is set to 1 and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None

Floating-Point Double-Precision Negate

\[ \text{efdneg } \text{RT}, \text{RA} \]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
| & 4 & \text{RT} & \text{RA} & /// & 742 \\
\hline
\end{array}
\]

\[ \text{RT}_{0:63} \leftarrow \neg \langle \text{RA} \rangle_{0} || \langle \text{RA} \rangle_{1:63} \]

The sign bit of register RA is complemented and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:
None
Floating-Point Double-Precision Add
EVX-form

efdadd RT,RA,RB

RT₀:63 ← (RA)₀:63 +dp (RB)₀:63
RA is added to RB and the result is stored in register RT.
If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:
FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Subtract
EVX-form

efdsub RT,RA,RB

RT₀:63 ← (RA)₀:63 -dp (RB)₀:63
RB is subtracted from RA and the result is stored in register RT.
If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:
FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Multiply
EVX-form

efdmul RT,RA,RB

RT₀:63 ← (RA)₀:63 ×dp (RB)₀:63
RA is multiplied by RB and the result is stored in register RT.

Special Registers Altered:
FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Divide
EVX-form

efddiv RT,RA,RB

RT₀:63 ← (RA)₀:63 ÷dp (RB)₀:63
RA is divided by RB and the result is stored in register RT.

Special Registers Altered:
FINV FINVS
FG FX FINXS
FDBZ FDBZS
FOVF FOVFS
FUNF FUNFS
### Floating-Point Double-Precision Compare Greater Than EVX-form

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>748</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{al} & \leftarrow (\text{RA})_{0:63} \\
\text{bl} & \leftarrow (\text{RB})_{0:63} \\
\text{if} (\text{al} > \text{bl}) & \text{ then } \text{cl} \leftarrow 1 \\
\text{else cl} & \leftarrow 0 \\
\text{CR}_{4 \times \text{BF}:4 \times \text{BF}+3} & \leftarrow \underline{\text{undefined}} || \text{cl} || \underline{\text{undefined}} || \underline{\text{undefined}}
\end{align*}
\]

RA is compared against RB. If RA is greater than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**
- FINV
- FINVS
- FG
- FX
- CR field BF

### Floating-Point Double-Precision Compare Equal EVX-form

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>750</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{al} & \leftarrow (\text{RA})_{0:63} \\
\text{bl} & \leftarrow (\text{RB})_{0:63} \\
\text{if} (\text{al} = \text{bl}) & \text{ then } \text{cl} \leftarrow 1 \\
\text{else cl} & \leftarrow 0 \\
\text{CR}_{4 \times \text{BF}:4 \times \text{BF}+3} & \leftarrow \underline{\text{undefined}} || \text{cl} || \underline{\text{undefined}} || \underline{\text{undefined}}
\end{align*}
\]

RA is compared against RB. If RA is equal to RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**
- FINV
- FINVS
- FG
- FX
- CR field BF

### Floating-Point Double-Precision Compare Less Than EVX-form

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>749</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{al} & \leftarrow (\text{RA})_{0:63} \\
\text{bl} & \leftarrow (\text{RB})_{0:63} \\
\text{if} (\text{al} < \text{bl}) & \text{ then } \text{cl} \leftarrow 1 \\
\text{else cl} & \leftarrow 0 \\
\text{CR}_{4 \times \text{BF}:4 \times \text{BF}+3} & \leftarrow \underline{\text{undefined}} || \text{cl} || \underline{\text{undefined}} || \underline{\text{undefined}}
\end{align*}
\]

RA is compared against RB. If RA is less than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of ‘e’ and ‘f’ directly.

**Special Registers Altered:**
- FINV
- FINVS
- FG
- FX
- CR field BF

### Floating-Point Double-Precision Test Greater Than EVX-form

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>764</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{al} & \leftarrow (\text{RA})_{0:63} \\
\text{bl} & \leftarrow (\text{RB})_{0:63} \\
\text{if} (\text{al} > \text{bl}) & \text{ then } \text{cl} \leftarrow 1 \\
\text{else cl} & \leftarrow 0 \\
\text{CR}_{4 \times \text{BF}:4 \times \text{BF}+3} & \leftarrow \underline{\text{undefined}} || \text{cl} || \underline{\text{undefined}} || \underline{\text{undefined}}
\end{align*}
\]

RA is compared against RB. If RA is greater than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are generated during the execution of `efdtstgt`.

**Special Registers Altered:**
- CR field BF

---

### Programming Note

In an implementation, the execution of `efdtstgt` is likely to be faster than the execution of `efdcmpgt`; however, if strict IEEE 754 compliance is required, the program should use `efdcmpgt`.

---

662  Power ISA™ - Book I
### Floating-Point Double-Precision Test Less Than

**EVX-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BF, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efdtstlt</td>
<td>BF, RA, RB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>765</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{aligned} 
& a_0 \leftarrow (R A)_{5:63} \\
& b_1 \leftarrow (R B)_{5:63} \\
& \text{if } (a_0 < b_1) \text{ then } c_l \leftarrow 1 \\
& \text{else } c_l \leftarrow 0 \\
& \text{CR}_{4 \times BF:4 \times BF+3} \leftarrow \text{undefined} || c_l || \text{undefined} || \text{undefined} \\
\end{aligned} \]

RA is compared against RB. If RA is less than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are generated during the execution of efdtstlt.

**Special Registers Altered:**
- CR field BF

**Programming Note**
In an implementation, the execution of efdtstlt is likely to be faster than the execution of efdcmplt; however, if strict IEEE 754 compliance is required, the program should use efdcmplt.

### Convert Floating-Point Double-Precision from Signed Integer

**EVX-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efdcfsi</td>
<td>RT, RB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>753</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ RT_{5:63} \leftarrow \text{CnvrtI32ToFp64}(I R B)_{32:63}, S, I \]

The signed integer low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

**Special Registers Altered:**
- None

### Floating-Point Double-Precision Test Equal

**EVX-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BF, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efdtsteq</td>
<td>BF, RA, RB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>BF</th>
<th>RA</th>
<th>RB</th>
<th>766</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{aligned} 
& a_0 \leftarrow (R A)_{5:63} \\
& b_1 \leftarrow (R B)_{5:63} \\
& \text{if } (a_0 = b_1) \text{ then } c_l \leftarrow 1 \\
& \text{else } c_l \leftarrow 0 \\
& \text{CR}_{4 \times BF:4 \times BF+3} \leftarrow \text{undefined} || c_l || \text{undefined} || \text{undefined} \\
\end{aligned} \]

RA is compared against RB. If RA is equal to RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘e’ and ‘f’ directly.

No exceptions are generated during the execution of efdtsteq.

**Special Registers Altered:**
- CR field BF

**Programming Note**
In an implementation, the execution of efdtsteq is likely to be faster than the execution of efdccmpeq; however, if strict IEEE 754 compliance is required, the program should use efdccmpeq.

### Convert Floating-Point Double-Precision from Unsigned Integer

**EVX-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>efdcfui</td>
<td>RT, RB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>752</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ RT_{5:63} \leftarrow \text{CnvrtI32ToFp64}(U R B)_{32:63}, U, I \]

The unsigned integer low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

**Special Registers Altered:**
- None
**Convert Floating-Point Double-Precision from Signed Integer Doubleword**

**EVX-form**

```
efdcsid RT,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>11</th>
<th>RB</th>
<th>739</th>
</tr>
</thead>
</table>

\[ RT_{0:63} \leftarrow \text{CnvtxI64ToFP64}((\text{RB})_{0:63}, \text{S}) \]

The signed integer doubleword in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

**Corequisite Categories:**

- 64-Bit

**Special Registers Altered:**

- FINXS
- FG
- FX

**Convert Floating-Point Double-Precision from Unsigned Integer Doubleword**

**EVX-form**

```
efdcluid RT,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>11</th>
<th>RB</th>
<th>738</th>
</tr>
</thead>
</table>

\[ RT_{0:63} \leftarrow \text{CnvtxI64ToFP64}((\text{RB})_{0:63}, \text{U}) \]

The unsigned integer doubleword in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

**Corequisite Categories:**

- 64-Bit

**Special Registers Altered:**

- FINXS
- FG
- FX

**Convert Floating-Point Double-Precision from Signed Fraction**

**EVX-form**

```
efdcsf RT,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>11</th>
<th>RB</th>
<th>755</th>
</tr>
</thead>
</table>

\[ RT_{0:63} \leftarrow \text{CnvtxI32ToFP64}((\text{RB})_{32:63}, \text{S}, \text{F}) \]

The signed fractional low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

**Special Registers Altered:**

- None

**Convert Floating-Point Double-Precision to Signed Integer**

**EVX-form**

```
efdctsi RT,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>11</th>
<th>RB</th>
<th>757</th>
</tr>
</thead>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtxFP64ToI32Sat}((\text{RB})_{0:63}, \text{S}, \text{RND}, \text{I}) \]

The double-precision floating-point value in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**

- FINV
- FINVS
- FINXS
- FG
- FX

**Convert Floating-Point Double-Precision from Unsigned Fraction**

**EVX-form**

```
efdctuf RT,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>11</th>
<th>RB</th>
<th>754</th>
</tr>
</thead>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtxFP64ToI32Sat}((\text{RB})_{0:63}, \text{U}, \text{RND}, \text{I}) \]

The unsigned fractional low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

**Special Registers Altered:**

- None

**Convert Floating-Point Double-Precision to Unsigned Integer**

**EVX-form**

```
efdctui RT,RB
```

<table>
<thead>
<tr>
<th>4</th>
<th>RT</th>
<th>11</th>
<th>RB</th>
<th>756</th>
</tr>
</thead>
</table>

\[ RT_{32:63} \leftarrow \text{CnvtxFP64ToI32Sat}((\text{RB})_{0:63}, \text{U}, \text{RND}, \text{I}) \]

The double-precision floating-point value in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Special Registers Altered:**

- FINV
- FINVS
- FINXS
- FG
- FX
Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero  
**EVX-form**

\[
\text{efdctsidz} \quad RT, RB
\]

The double-precision floating-point value in register RB is converted to a signed integer doubleword using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 64-bit integer. NaNs are converted as though they were zero.

**Corequisite Categories:**
- 64-Bit

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

---

Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero  
**EVX-form**

\[
\text{efdctuidz} \quad RT, RB
\]

The double-precision floating-point value in register RB is converted to an unsigned integer doubleword using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 64-bit integer. NaNs are converted as though they were zero.

**Corequisite Categories:**
- 64-Bit

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX
**Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero**

**EVX-form**

efdctsiz  RT, RB

```
  4  6  11  16  21  762
```

The double-precision floating-point value in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Corequisite Categories:**
- SPE.Embedded Float Scalar Single or SPE.Embedded Float Vector

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

---

**Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero**

**EVX-form**

efdctuiz  RT, RB

```
  4  6  11  16  21  760
```

The double-precision floating-point value in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

**Corequisite Categories:**
- SPE.Embedded Float Scalar Single or SPE.Embedded Float Vector

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

---

**Convert Floating-Point Double-Precision to Signed Fraction**

**EVX-form**

efdctsf  RT, RB

```
  4  6  11  16  21  759
```

The double-precision floating-point value in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

**Corequisite Categories:**
- SPE.Embedded Float Scalar Single or SPE.Embedded Float Vector

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

---

**Convert Floating-Point Double-Precision to Unsigned Fraction**

**EVX-form**

efdctuf  RT, RB

```
  4  6  11  16  21  758
```

The double-precision floating-point value in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

**Corequisite Categories:**
- SPE.Embedded Float Scalar Single or SPE.Embedded Float Vector

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX

---

**Floating-Point Double-Precision Convert from Single-Precision**

**EVX-form**

efdcs  RT, RB

```
  4  6  11  16  21  751
```

The single-precision floating-point value in the low element of register RB is converted to a double-precision floating-point value and the result is placed in register RT.

**Corequisite Categories:**
- SPE.Embedded Float scalar Single or SPE.Embedded Float Vector

**Special Registers Altered:**
- FINV
- FINVS
- FINXS
- FG
- FX
Floating-Point Single-Precision Convert
from Double-Precision EVX-form

```
efscfd RT,RB

FP64format f;
FP32format result;
f ← (RB)[0:63]
if (fexp = 0) & (ffrac = 0)) then
    result ← fsign || 310
else if Is64NaNorInfinity(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 0b11111110 || 231
else if Is64Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 310
else
    unbias ← fexp - 1023
    if unbias > 127 then
        result ← fsign || 0b11111110 || 231
        SPEFSCR_FOVF ← 1
    else if unbias < -126 then
        result ← fsign || 310
        SPEFSCR_FUNF ← 1
    else
        resultsign ← fsign
        resultexp ← unbias + 127
        resultfrac ← ffrac[0:22]
guard ← ffrac[23]
sticky ← (ffrac[24:51] ≠ 0)
result ← Round32(result, LO, guard, sticky)

SPEFSCR_FG ← guard
SPEFSCR_FX ← sticky
if guard | sticky then
    SPEFSCR_FINXS ← 1

RT[32:63] ← result
```

The double-precision floating-point value in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Corequisite Categories:
SPE.Embedded Float Scalar Scalar

Special Registers Altered:
FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS
9.4 Embedded Floating-Point Results Summary

The following tables summarize the results of various types of Embedded Floating-Point operations on various combinations of input operands. Flag settings are performed on appropriate element flags. For all the tables the following annotation and general rules apply:

- "*" denotes that this status flag is set based on the results of the calculation.
- "Calc_" denotes that the result is updated with the results of the computation.
- "max" denotes the maximum normalized number with the sign set to the computation [sign(operand A) XOR sign(operand B)].
- "amax" denotes the maximum normalized number with the sign set to the sign of Operand A.
- "bmax" denotes the maximum normalized number with the sign set to the sign of Operand B.
- "pmax" denotes the maximum normalized positive number. The encoding for single-precision is: 0x7F7FFFFF. The encoding for double-precision is: 0x7FEFFFFF_FFFFFFFF.
- "nmax" denotes the maximum normalized negative number. The encoding for single-precision is: 0xFF7FFFFF. The encoding for double-precision is: 0xFFEFFFFF_FFFFFFFF.
- "pmin" denotes the minimum normalized positive number. The encoding for single-precision is: 0x00800000. The encoding for double-precision is: 0x00100000_00000000.
- "nmin" denotes the minimum normalized negative number. The encoding for single-precision is: 0x80800000. The encoding for double-precision is: 0x80100000_00000000.
- Calculations that overflow or underflow saturate. Overflow for operations that have a floating-point result force the result to "max". Underflow for operations that have a floating-point result force the result to zero. Overflow for operations that have a signed integer result force the result to 0x7FFFFFFF (positive) or 0x80000000 (negative). Underflow for operations that have an unsigned integer result force the result to 0xFFFFFFF (positive) or 0x00000000 (negative).
- 1 (superscript) denotes that the sign of the result is positive when the sign of Operand A and the sign of Operand B are different, for all rounding modes except round to -infinity, where the sign of the result is then negative.
- 2 (superscript) denotes that the sign of the result is positive when the sign of Operand A and the sign of Operand B are the same, for all rounding modes except round to -infinity, where the sign of the result is then negative.
- 3 (superscript) denotes that the sign for any multiply or divide is always the result of the operation [sign(Operand A) XOR sign(Operand B)].
- 4 (superscript) denotes that if an overflow is detected, the result may be saturated.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>∞</td>
<td>∞</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>∞</td>
<td>NaN</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>∞</td>
<td>denorm</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>∞</td>
<td>zero</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>∞</td>
<td>Norm</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>NaN</td>
<td>∞</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>NaN</td>
<td>NaN</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>NaN</td>
<td>denorm</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>NaN</td>
<td>zero</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>NaN</td>
<td>norm</td>
<td>amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>denorm</td>
<td>∞</td>
<td>bmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>denorm</td>
<td>NaN</td>
<td>bmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>denorm</td>
<td>denorm</td>
<td>zero³</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>denorm</td>
<td>zero</td>
<td>zero³</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>denorm</td>
<td>norm</td>
<td>operand_b⁴</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>zero</td>
<td>∞</td>
<td>bmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>zero</td>
<td>NaN</td>
<td>bmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>zero</td>
<td>denorm</td>
<td>zero³</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Operation</td>
<td>Operand A</td>
<td>Operand B</td>
<td>Result</td>
<td>FINV</td>
<td>FOVF</td>
<td>FUNF</td>
<td>FDBZ</td>
<td>FINX</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>Add</td>
<td>zero</td>
<td>zero</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>zero</td>
<td>norm operand_b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>norm</td>
<td>infinity</td>
<td>bmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>norm</td>
<td>NaN</td>
<td>bmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>norm</td>
<td>denorm operand_a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>norm</td>
<td>zero operand_a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add</td>
<td>norm</td>
<td>norm <em>Calc</em></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Subtract**

| Sub       | infinity | infinity | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | infinity | NaN      | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | infinity | denorm  | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | infinity | zero    | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | infinity | Norm    | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | NaN      | infinity | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | NaN      | NaN      | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | NaN      | denorm  | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | NaN      | zero    | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | NaN      | norm    | amax   | 1    | 0    | 0    | 0    | 0    |
| Sub       | denorm   | infinity | -bmax  | 1    | 0    | 0    | 0    | 0    |
| Sub       | denorm   | NaN      | -bmax  | 1    | 0    | 0    | 0    | 0    |
| Sub       | denorm   | denorm zero^2 | 1    | 0    | 0    | 0    | 0    | 0    |
| Sub       | denorm   | zero zero^2 | 1    | 0    | 0    | 0    | 0    | 0    |
| Sub       | denorm   | norm -operand_b^4 | 1    | 0    | 0    | 0    | 0    | 0    |
| Sub       | zero     | infinity | -bmax  | 1    | 0    | 0    | 0    | 0    |
| Sub       | zero     | NaN      | -bmax  | 1    | 0    | 0    | 0    | 0    |
| Sub       | zero     | denorm zero^2 | 1    | 0    | 0    | 0    | 0    | 0    |
| Sub       | zero     | norm -operand_b^4 | 0    | 0    | 0    | 0    | 0    | 0    |
| Sub       | norm     | infinity | -bmax  | 1    | 0    | 0    | 0    | 0    |
| Sub       | norm     | NaN      | -bmax  | 1    | 0    | 0    | 0    | 0    |
| Sub       | norm     | denorm operand_a^4 | 1    | 0    | 0    | 0    | 0    | 0    |
| Sub       | norm     | zero operand_a^4 | 0    | 0    | 0    | 0    | 0    | 0    |
| Sub       | norm     | norm _Calc_ | 0    | 0    | 0    | 0    | 0    | 0    |

- **Multiply**

| Mul       | infinity | infinity | max    | 1    | 0    | 0    | 0    | 0    |
| Mul       | infinity | NaN      | max    | 1    | 0    | 0    | 0    | 0    |
| Mul       | infinity | denorm  | zero   | 1    | 0    | 0    | 0    | 0    |
| Mul       | infinity | zero    | zero   | 1    | 0    | 0    | 0    | 0    |
| Mul       | infinity | Norm    | max    | 1    | 0    | 0    | 0    | 0    |
| Mul       | NaN      | infinity | max    | 1    | 0    | 0    | 0    | 0    |
| Mul       | NaN      | NaN      | max    | 1    | 0    | 0    | 0    | 0    |
| Mul       | NaN      | denorm  | zero   | 1    | 0    | 0    | 0    | 0    |
| Mul       | NaN      | zero    | zero   | 1    | 0    | 0    | 0    | 0    |
| Mul       | NaN      | norm    | max    | 1    | 0    | 0    | 0    | 0    |

Table 115: Embedded Floating-Point Results Summary—Add, Sub, Mul (Continued)
Table 115: Embedded Floating-Point Results Summary—Add, Sub, Mul, Div (Continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul</td>
<td>denorm</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>denorm</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>denorm</td>
<td>denorm</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>denorm</td>
<td>zero</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>denorm</td>
<td>norm</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>zero</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>zero</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>zero</td>
<td>denorm</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>zero</td>
<td>zero</td>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>zero</td>
<td>norm</td>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>norm</td>
<td>∞</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>norm</td>
<td>NaN</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>norm</td>
<td>denorm</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>norm</td>
<td>zero</td>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mul</td>
<td>norm</td>
<td>norm</td>
<td>_Calc</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>Divide^3</td>
<td>∞</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>∞</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>∞</td>
<td>denorm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>∞</td>
<td>zero</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>NaN</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>NaN</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>NaN</td>
<td>denorm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>NaN</td>
<td>zero</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>NaN</td>
<td>norm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>denorm</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>denorm</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>denorm</td>
<td>denorm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>denorm</td>
<td>zero</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>denorm</td>
<td>norm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>zero</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>zero</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>zero</td>
<td>denorm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>zero</td>
<td>zero</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>zero</td>
<td>norm</td>
<td>max</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>norm</td>
<td>∞</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>norm</td>
<td>NaN</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>norm</td>
<td>denorm</td>
<td>max</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>norm</td>
<td>zero</td>
<td>max</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divide^3</td>
<td>norm</td>
<td>norm</td>
<td>_Calc</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>
Table 116: Embedded Floating-Point Results Summary—Single Convert from Double

<table>
<thead>
<tr>
<th>Operand B</th>
<th>efscfd result</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞</td>
<td>pmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-∞</td>
<td>nmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+NaN</td>
<td>pmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-NaN</td>
<td>nmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+denorm</td>
<td>+zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-denorm</td>
<td>-zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+zero</td>
<td>+zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-zero</td>
<td>-zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>norm</td>
<td><em>Calc</em></td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 117: Embedded Floating-Point Results Summary—Double Convert from Single

<table>
<thead>
<tr>
<th>Operand B</th>
<th>edfdfs result</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞</td>
<td>pmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-∞</td>
<td>nmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+NaN</td>
<td>pmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-NaN</td>
<td>nmax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+denorm</td>
<td>+zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-denorm</td>
<td>-zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+zero</td>
<td>+zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-zero</td>
<td>-zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>norm</td>
<td><em>Calc</em></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 118: Embedded Floating-Point Results Summary—Convert to Unsigned

<table>
<thead>
<tr>
<th>Operand B</th>
<th>Integer Result ctui[d][z]</th>
<th>Fractional Result ctuf</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞</td>
<td>0xFFFF_FFFF</td>
<td>0x7FFF_FFFF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-∞</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+NaN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-NaN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>denorm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+norm</td>
<td><em>Calc</em></td>
<td><em>Calc</em></td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>-norm</td>
<td><em>Calc</em></td>
<td><em>Calc</em></td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>
### Table 119: Embedded Floating-Point Results Summary—Convert to Signed

<table>
<thead>
<tr>
<th>Operand</th>
<th>Integer Result [d][z]</th>
<th>Fractional Result ctsf</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞</td>
<td>0x7FFF_FFFF</td>
<td>0x7FFF_FFFF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-∞</td>
<td>0x8000_0000</td>
<td>0x8000_0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+NaN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-NaN</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>denorm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+norm</td>
<td><em>Calc</em></td>
<td><em>Calc</em></td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>-norm</td>
<td><em>Calc</em></td>
<td><em>Calc</em></td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

### Table 120: Embedded Floating-Point Results Summary—Convert from Unsigned

<table>
<thead>
<tr>
<th>Operand</th>
<th>Integer Source cfui</th>
<th>Fractional Source cfuf</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>zero</td>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>norm</td>
<td><em>Calc</em></td>
<td><em>Calc</em></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

### Table 121: Embedded Floating-Point Results Summary—Convert from Signed

<table>
<thead>
<tr>
<th>Operand</th>
<th>Integer Source cfsi</th>
<th>Fractional Source cfsf</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>zero</td>
<td>zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>norm</td>
<td><em>Calc</em></td>
<td><em>Calc</em></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

### Table 122: Embedded Floating-Point Results Summary—*abs, *nabs, *neg

<table>
<thead>
<tr>
<th>Operand</th>
<th>*abs</th>
<th>*nabs</th>
<th>*neg</th>
<th>FINV</th>
<th>FOVF</th>
<th>FUNF</th>
<th>FDBZ</th>
<th>FINX</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞</td>
<td>pmax</td>
<td>+∞</td>
<td>-amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-∞</td>
<td>pmax</td>
<td>+∞</td>
<td>-amax</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+NaN</td>
<td>pmax</td>
<td>NaN</td>
<td>-NaN</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-NaN</td>
<td>pmax</td>
<td>NaN</td>
<td>-NaN</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+denorm</td>
<td>+zero</td>
<td>+denorm</td>
<td>+zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-denorm</td>
<td>+zero</td>
<td>+denorm</td>
<td>+zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+zero</td>
<td>+zero</td>
<td>-zero</td>
<td>-zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-zero</td>
<td>+zero</td>
<td>-zero</td>
<td>+zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+norm</td>
<td>+norm</td>
<td>-norm</td>
<td>-norm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-norm</td>
<td>+norm</td>
<td>-norm</td>
<td>+norm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 10. Legacy Move Assist Instruction
[Category: Legacy Move Assist]

Determine Leftmost Zero Byte  

\[ \text{dlmzb} \quad RA, RS, RB \quad (Rc=0) \]
\[ \text{dlmzb.} \quad RA, RS, RB \quad (Rc=1) \]

\[
\begin{array}{cccccc}
\text{do while } (x<8) \& (y=0) \\
\text{x} & \leftarrow & x + 1 \\
\text{if } d_i+32:39 \text{ } = \text{ } 0 & \text{then} \\
\text{y} & \leftarrow & 1 \\
\text{else} \\
\text{i} & \leftarrow & i + 8 \\
\text{x} & \leftarrow & (RS)_{32:63} || (RB)_{32:63} \\
\text{y} & \leftarrow & 0 \\
\text{enddo} & \quad \text{if } \text{Rc} = \text{1} & \text{do} & \quad \text{else} \\
\text{i} & \leftarrow & 0 \\
\text{x} & \leftarrow & x \\
\text{XER}_{57:63} & \leftarrow & x \\
\text{if } \text{Rc} = \text{1} & \text{then} & \text{CR}_{35} & \leftarrow & \text{SO} & \quad \text{if } \text{y} = \text{1} & \text{then} & \text{if } \text{x} < \text{5} & \text{then} & \text{CR}_{32:34} & \leftarrow & 0b010 \\
\text{else} & \quad \text{else} & \text{CR}_{32:34} & \leftarrow & 0b100 \\
\text{else} & \quad \text{else} & \text{CR}_{32:34} & \leftarrow & 0b001 \\
\end{array}
\]

The contents of bits 32:63 of register RS and the contents of bits 32:63 of register RB are concatenated to form an 8-byte operand. The operand is searched for the leftmost byte in which each bit is 0 (i.e., a null byte).

Bytes in the operand are numbered from left to right starting with 1. If a null byte is found, its byte number is placed into bits 57:63 of the XER and into register RA. Otherwise, the value \(0b000 \ 1000\) is placed into both bits 57:63 of the XER and register RA.

If Rc is equal to 1, SO is copied into bit 35 of the CR and bits 32:34 of the CR are updated as follows:

- If no null byte is found, bits 32:34 of the CR are set to \(0b001\).
- If the leftmost null byte is in the first 4 bytes (i.e., from register RS), bits 32:34 of the CR are set to \(0b010\).
- If the leftmost null byte is in the last 4 bytes (i.e., from register RB), bits 32:34 of the CR are set to \(0b100\).
Chapter 11. Legacy Integer Multiply-Accumulate Instructions [Category: Legacy Integer Multiply-Accumulate]

The Legacy Integer Multiply-Accumulate instructions with Rc=1 set the first three bits of CR Field 0 based on the 32-bit result, as described in Section 3.3.8, “Other Fixed-Point Instructions”.

The XO-form Legacy Integer Multiply-Accumulate instructions set SO and OV when OE=1 to reflect overflow of the 32-bit result.

### Multiply Accumulate Cross Halfword to Word Modulo Signed

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>172</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

macchw RT,RA,RB (OE=0 Rc=0)
machch. RT,RA,RB (OE=0 Rc=1)
machchwo RT,RA,RB (OE=1 Rc=0)
machchwo. RT,RA,RB (OE=1 Rc=1)

\[
\begin{align*}
\text{prod}_{0:31} & \leftarrow (\text{RA})_{48:63} \times_{\text{si}} (\text{RB})_{32:47} \\
\text{temp}_{0:32} & \leftarrow \text{prod}_{0:31} + (\text{RT})_{32:63} \\
\text{RT}_{32:63} & \leftarrow \text{temp}_{1:32} \\
\text{RT}_{0:31} & \leftarrow \text{undefined}
\end{align*}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**
- SO OV (if OE=1)
- CR0 (if Rc=1)

### Multiply Accumulate Cross Halfword to Word Saturate Signed

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>236</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>22</td>
<td>31</td>
</tr>
</tbody>
</table>

macchws RT,RA,RB (OE=0 Rc=0)
machchws. RT,RA,RB (OE=0 Rc=1)
machchwso RT,RA,RB (OE=1 Rc=0)
machchwso. RT,RA,RB (OE=1 Rc=1)

\[
\begin{align*}
\text{prod}_{0:31} & \leftarrow (\text{RA})_{48:63} \times_{\text{si}} (\text{RB})_{32:47} \\
\text{temp}_{0:32} & \leftarrow \text{prod}_{0:31} + (\text{RT})_{32:63} \\
\text{if temp} & < -2^{31} \quad \text{then} \quad \text{RT}_{32:63} \leftarrow 0x8000_0000 \\
\text{else if temp} & > 2^{31}-1 \quad \text{then} \quad \text{RT}_{32:63} \leftarrow 0x7FFF_FFFF \\
\text{else} & \quad \text{RT}_{32:63} \leftarrow \text{temp}_{1:32} \\
\text{RT}_{0:31} & \leftarrow \text{undefined}
\end{align*}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the sum is greater than 2^{31}-1, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**
- SO OV (if OE=1)
- CR0 (if Rc=1)

Programming Note

Notice that CR Field 0 may not reflect the “true” (infinitely precise) result if overflow occurs.
### Multiply Accumulate Cross Halfword to Word Modulo Unsigned XO-form

<table>
<thead>
<tr>
<th>prod0:31</th>
<th>temp0:32</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>140</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RA)48:63 \times_{ui} (RB)32:47</td>
<td>prod0:31 + (RT)32:63</td>
<td>RT</td>
<td>temp1:32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

- SO
- OV (if OE=1)
- CR0 (if Rc=1)

### Multiply Accumulate Cross Halfword to Word Saturate Unsigned XO-form

<table>
<thead>
<tr>
<th>prod0:31</th>
<th>temp0:32</th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>204</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RA)48:63 \times_{ui} (RB)32:47</td>
<td>prod0:31 + (RT)32:63</td>
<td>if temp &gt; 2^{32}-1 then RT \leftarrow 0xFFFF_FFFF</td>
<td>else</td>
<td>RT \leftarrow temp1:32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than $2^{32}-1$, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

- SO
- OV (if OE=1)
- CR0 (if Rc=1)
Multiply Accumulate High Halfword to Word Modulo Signed XO-form

machhw   RT,RA,RB   (OE=0 Rc=0)
machhw.  RT,RA,RB   (OE=0 Rc=1)
machhwso RT,RA,RB   (OE=1 Rc=0)
machhwso. RT,RA,RB  (OE=1 Rc=1)

\[
\begin{array}{cccccc}
\text{prod}_{0:31} & \leftarrow & (RA)_{32:47} \times_{\text{si}} (RB)_{32:47} \\
\text{temp}_{0:32} & \leftarrow & \text{prod}_{0:31} + (RT)_{32:63} \\
RT_{32:63} & \leftarrow & \text{temp}_{1:32} \\
\end{array}
\]

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

- SO
- OV
- CR0 (if OE=1)
- CR0 (if Rc=1)

Multiply Accumulate High Halfword to Word Saturate Signed XO-form

machhws RT,RA,RB   (OE=0 Rc=0)
machhws. RT,RA,RB   (OE=0 Rc=1)
machhwsO RT,RA,RB   (OE=1 Rc=0)
machhwsO. RT,RA,RB  (OE=1 Rc=1)

\[
\begin{array}{cccccc}
\text{prod}_{0:31} & \leftarrow & (RA)_{32:47} \times_{\text{si}} (RB)_{32:47} \\
\text{temp}_{0:32} & \leftarrow & \text{prod}_{0:31} + (RT)_{32:63} \\
\text{if} \text{ temp} < -2^{31} \text{ then } RT_{32:63} & \leftarrow & 0x8000_0000 \\
\text{else if} \text{ temp} > 2^{31}-1 \text{ then } RT_{32:63} & \leftarrow & 0x7FFF_FFFF \\
\text{else} & \text{ temp}_{1:32} & \leftarrow & RT_{32:63} \\
\text{RT}_{32:63} & \leftarrow & \text{undefined} \\
\end{array}
\]

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than \(-2^{31}\), then the value \(0x8000_0000\) is placed into bits 32:63 of register RT.

If the sum is greater than \(2^{31}-1\), then the value \(0x7FFF_FFFF\) is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

- SO
- OV
- CR0 (if OE=1)
Multiply Accumulate High Halfword to Word Modulo Unsigned XO-form

machhwu RT,RA,RB (OE=0 Rc=0)
machhwu. RT,RA,RB (OE=0 Rc=1)
machhwuo RT,RA,RB (OE=1 Rc=0)
machhwuo. RT,RA,RB (OE=1 Rc=1)

prod0:31 (RA)32:47 ×ui (RB)32:47

temp0:32 prod0:31 + (RT)32:63

RT32:63 temp1:32

RT0:31 undefined

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate High Halfword to Word Saturate Unsigned XO-form

machhwsu RT,RA,RB (OE=0 Rc=0)
machhwsu. RT,RA,RB (OE=0 Rc=1)
machhwsuo RT,RA,RB (OE=1 Rc=0)
machhwsuo. RT,RA,RB (OE=1 Rc=1)

prod0:31 (RA)32:47 ×ui (RB)32:47

if temp > 2^32-1 then RT ← 0xFFFF_FFFF
else RT ← temp1:32

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^32-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)
Multiply Accumulate Low Halfword to Word Modulo Signed XO-form

\[
\text{maclhw} \quad \text{RT,RA,RB} \quad (OE=0 \quad \text{Rc}=0) \\
\text{maclhwo} \quad \text{RT,RA,RB} \quad (OE=1 \quad \text{Rc}=0) \\
\text{maclhwo.} \quad \text{RT,RA,RB} \quad (OE=1 \quad \text{Rc}=1)
\]

\[
\begin{array}{cccccc}
\text{4} & \text{6} & \text{11} & \text{16} & \text{OE} & \text{RC} \\
0 & \text{RT} & \text{RA} & \text{RB} & 21 & 22 \\
\end{array}
\]

\[
\begin{align*}
\text{prod}_{0:31} & \leftarrow (\text{RA})_{48:63} \times_{\text{si}} (\text{RB})_{48:63} \\
\text{temp}_{0:32} & \leftarrow \text{prod}_{0:31} + (\text{RT})_{32:63} \\
\text{RT}_{32:63} & \leftarrow \text{temp}_{1:32} \\
\text{RT}_{0:31} & \leftarrow \text{undefined}
\end{align*}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
- SO
- OV (if OE=1)
- CR0 (if Rc=1)

Multiply Accumulate Low Halfword to Word Saturate Signed XO-form

\[
\text{maclhws} \quad \text{RT,RA,RB} \quad (OE=0 \quad \text{Rc}=0) \\
\text{maclhws.} \quad \text{RT,RA,RB} \quad (OE=0 \quad \text{Rc}=1) \\
\text{maclhwso} \quad \text{RT,RA,RB} \quad (OE=1 \quad \text{Rc}=0) \\
\text{maclhwso.} \quad \text{RT,RA,RB} \quad (OE=1 \quad \text{Rc}=1)
\]

\[
\begin{array}{cccccc}
\text{4} & \text{6} & \text{11} & \text{16} & \text{OE} & \text{RC} \\
0 & \text{RT} & \text{RA} & \text{RB} & 21 & 22 \\
\end{array}
\]

\[
\begin{align*}
\text{prod}_{0:31} & \leftarrow (\text{RA})_{48:63} \times_{\text{si}} (\text{RB})_{48:63} \\
\text{temp}_{0:32} & \leftarrow \text{prod}_{0:31} + (\text{RT})_{32:63} \\
\text{if temp} \prec -2^{31} & \text{ then } \text{RT}_{32:63} \leftarrow 0x8000_0000 \\
\text{else if temp} \succ 2^{31}-1 & \text{ then } \text{RT}_{32:63} \leftarrow 0x7FFF_FFFF \\
\text{else} & \text{ RT}_{32:63} \leftarrow \text{temp}_{1:32} \\
\text{RT}_{0:31} & \leftarrow \text{undefined}
\end{align*}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than $-2^{31}$, then the value $0x8000_0000$ is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value $0x7FFF_FFFF$ is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
- SO
- OV (if OE=1)
- CR0 (if Rc=1)
Multiply Accumulate Low Halfword to Word Modulo Unsigned XO-form

```
maclhwu RT, RA, RB (OE=0 Rc=0)
maclhwu. RT, RA, RB (OE=0 Rc=1)
maclhwuo RT, RA, RB (OE=1 Rc=0)
maclhwuo. RT, RA, RB (OE=1 Rc=1)
```

```
prod0:31 ← (RA)48:63 ×ui (RB)48:63
temp0:32 ← prod0:31 + (RT)32:63
RT32:63 ← temp1:32
RT0:31 ← undefined
```

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**
- SO
- OV
- (if OE=1)
- CR0
- (if Rc=1)

Multiply Accumulate Low Halfword to Word Saturate Unsigned XO-form

```
maclhwsu RT, RA, RB (OE=0 Rc=0)
maclhwsu. RT, RA, RB (OE=0 Rc=1)
maclhwsuo RT, RA, RB (OE=1 Rc=0)
maclhwsuo. RT, RA, RB (OE=1 Rc=1)
```

```
prod0:31 ← (RA)48:63 ×ui (RB)48:63
temp0:32 ← prod0:31 + (RT)32:63
if temp > 232-1 then RT ← 0xFFFF_FFFF
else
RT ← temp1:32
```

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**
- SO
- OV
- (if OE=1)
- CR0
- (if Rc=1)

Multiply Cross Halfword to Word Signed X-form

```
mulchw RT, RA, RB (Rc=0)
mulchwu. RT, RA, RB (Rc=1)
```

```
RT32:63 ← (RA)48:63 ×si (RB)32:47
RT0:31 ← undefined
```

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**
- CR0
- (if Rc=1)

Multiply Cross Halfword to Word Unsigned X-form

```
mulchwu RT, RA, RB (Rc=0)
mulchwu. RT, RA, RB (Rc=1)
```

```
RT32:63 ← (RA)48:63 ×ui (RB)32:47
RT0:31 ← undefined
```

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**
- CR0
- (if Rc=1)
### Multiply High Halfword to Word Signed

**X-form**

<table>
<thead>
<tr>
<th>mulhhw</th>
<th>RT, RA, RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhhw.</td>
<td>RT, RA, RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

$$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$$

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

CR0 (if Rc=1)

### Multiply High Halfword to Word Unsigned

**X-form**

<table>
<thead>
<tr>
<th>mulhhwu</th>
<th>RT, RA, RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhhwu.</td>
<td>RT, RA, RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

$$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$$

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

CR0 (if Rc=1)

### Multiply Low Halfword to Word Signed

**X-form**

<table>
<thead>
<tr>
<th>mullhw</th>
<th>RT, RA, RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mullhw.</td>
<td>RT, RA, RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

$$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$$

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

CR0 (if Rc=1)

### Multiply Low Halfword to Word Unsigned

**X-form**

<table>
<thead>
<tr>
<th>mullhwu</th>
<th>RT, RA, RB</th>
<th>(Rc=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mullhwu.</td>
<td>RT, RA, RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

$$RT_{32:63} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

CR0 (if Rc=1)
**Negative Multiply Accumulate Cross Halfword to Word Modulo Signed**

**XO-form**

\[
\text{nmacchw} \quad \text{RT,RA,RB} \quad \text{(OE}=0 \text{ Rc}=0) \\
\text{nmacchw.} \quad \text{RT,RA,RB} \quad \text{(OE}=0 \text{ Rc}=1) \\
\text{nmacchwo} \quad \text{RT,RA,RB} \quad \text{(OE}=1 \text{ Rc}=0) \\
\text{nmacchwo.} \quad \text{RT,RA,RB} \quad \text{(OE}=1 \text{ Rc}=1) \\
\]

\[
\begin{array}{ccccccc}
\text{prod}_{0:31} & \leftarrow & (\text{RA})_{48:63} \times_{\text{si}} (\text{RB})_{32:47} \\
\text{temp}_{0:32} & \leftarrow & (\text{RT})_{32:63} -_{\text{si}} \text{prod}_{0:31} \\
\text{RT}_{32:63} & \leftarrow & \text{temp}_{1:32} \\
\text{RT}_{0:31} & \leftarrow & \text{undefined} \\
\end{array}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

- SO
- OV
- CR0

**Negative Multiply Accumulate Cross Halfword to Word Saturate Signed**

**XO-form**

\[
\text{nmacchws} \quad \text{RT,RA,RB} \quad \text{(OE}=0 \text{ Rc}=0) \\
\text{nmacchws.} \quad \text{RT,RA,RB} \quad \text{(OE}=0 \text{ Rc}=1) \\
\text{nmacchwso} \quad \text{RT,RA,RB} \quad \text{(OE}=1 \text{ Rc}=0) \\
\text{nmacchwso.} \quad \text{RT,RA,RB} \quad \text{(OE}=1 \text{ Rc}=1) \\
\]

\[
\begin{array}{ccccccc}
\text{prod}_{0:31} & \leftarrow & (\text{RA})_{48:63} \times_{\text{si}} (\text{RB})_{32:47} \\
\text{temp}_{0:32} & \leftarrow & (\text{RT})_{32:63} -_{\text{si}} \text{prod}_{0:31} \\
\text{if temp} & < & -2^{31} & \text{then} & \text{RT}_{32:63} & \leftarrow & 0x8000_0000 \\
\text{if temp} & > & 2^{31}-1 & \text{then} & \text{RT}_{32:63} & \leftarrow & 0x7FFF_FFFF \\
\text{else} & & & \text{else} & \text{RT}_{32:63} & \leftarrow & \text{temp}_{1:32} \\
\text{RT}_{0:31} & \leftarrow & \text{undefined} \\
\end{array}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than \(-2^{31}\), then the value \(0x8000_0000\) is placed into bits 32:63 of register RT.

If the difference is greater than \(2^{31}-1\), then the value \(0x7FFF_FFFF\) is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

- SO
- OV
- CR0
Negative Multiply Accumulate High
Halfword to Word Modulo Signed

**XO-form**

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>46</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod0:31</td>
<td>(RA)32:47 × (RB)32:47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>temp0:32</td>
<td>(RT)32:63 - prod0:31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT32:63</td>
<td>temp1:32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>undefined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

- SO
- OV
- CR0

Negative Multiply Accumulate High
Halfword to Word Saturate Signed

**XO-form**

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
<th>OE</th>
<th>110</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod0:31</td>
<td>(RA)32:47 × (RB)32:47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>temp0:32</td>
<td>(RT)32:63 - prod0:31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if temp &lt; -231 then RT32:63 ← 0x8000_0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if temp &gt; 231-1 then RT32:63 ← 0x7FFF_FFFF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else RT32:63 ← temp1:32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT0:31</td>
<td>undefined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -231, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the difference is greater than 231-1, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

- SO
- OV
- CR0
**Negative Multiply Accumulate Low**  
**Halfword to Word Modulo Signed**  

**XO-form**

\[
\text{nmaclhw} \quad \text{RT,RA,RB} \quad \text{(OE=0 Rc=0)}
\]

\[
\text{nmaclhw} \quad \text{RT,RA,RB} \quad \text{(OE=0 Rc=1)}
\]

\[
\text{nmaclhwo} \quad \text{RT,RA,RB} \quad \text{(OE=1 Rc=0)}
\]

\[
\text{nmaclhwo} \quad \text{RT,RA,RB} \quad \text{(OE=1 Rc=1)}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
\text{prod}_{0:31} & \text{temp}_{0:32} & \text{RT}_{32:63} & \text{RT}_{0:31} \\
\hline
\langle RA \rangle_{48:63} \times_{\text{si}} \langle RB \rangle_{48:63} & \text{temp}_{0:32} & \text{RT}_{32:63} & \text{temp}_{1:32} & \text{RT}_{0:31} & \text{undefined} \\
\hline
\end{array}
\]

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

\[
\begin{array}{c|c}
\hline
\text{SO} & \text{OV} \\
\hline
\text{(if OE=1)} & \text{(if RC=1)} \\
\hline
\end{array}
\]

---

**Negative Multiply Accumulate Low**  
**Halfword to Word Saturate Signed**  

**XO-form**

\[
\text{nmaclhws} \quad \text{RT,RA,RB} \quad \text{(OE=0 Rc=0)}
\]

\[
\text{nmaclhws} \quad \text{RT,RA,RB} \quad \text{(OE=0 Rc=1)}
\]

\[
\text{nmaclhwso} \quad \text{RT,RA,RB} \quad \text{(OE=1 Rc=0)}
\]

\[
\text{nmaclhwso} \quad \text{RT,RA,RB} \quad \text{(OE=1 Rc=1)}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
\text{prod}_{0:31} & \text{temp}_{0:32} & \text{RT}_{32:63} & \text{RT}_{0:31} \\
\hline
\langle RA \rangle_{48:63} \times_{\text{si}} \langle RB \rangle_{48:63} & \text{temp}_{0:32} & \text{RT}_{32:63} & \text{temp}_{1:32} & \text{RT}_{0:31} & \text{undefined} \\
\hline
\end{array}
\]

If the difference is less than \(-2^{31}\), then the value \(0x8000_0000\) is placed into bits 32:63 of register RT.

If the difference is greater than \(2^{31}-1\), then the value \(0x7FFF_FFFF\) is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

**Special Registers Altered:**

\[
\begin{array}{c|c}
\hline
\text{SO} & \text{OV} \\
\hline
\text{(if OE=1)} & \text{(if RC=1)} \\
\hline
\end{array}
\]
Appendix A. Suggested Floating-Point Models
[Category: Floating-Point]

A.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the *Floating Round to Single-Precision* instruction.

\[
\begin{align*}
\text{If } (\text{FRB})_{1:11} < 897 \text{ and } (\text{FRB})_{1:63} > 0 \text{ then} \\
\quad \text{Do} \\
\quad \quad \text{If } \text{FPSCR}_{UE} = 0 \text{ then goto Disabled Exponent Underflow} \\
\quad \quad \text{If } \text{FPSCR}_{UE} = 1 \text{ then goto Enabled Exponent Underflow} \\
\quad \text{End} \\
\text{If } (\text{FRB})_{1:11} > 1150 \text{ and } (\text{FRB})_{1:11} < 2047 \text{ then} \\
\quad \text{Do} \\
\quad \quad \text{If } \text{FPSCR}_{OE} = 0 \text{ then goto Disabled Exponent Overflow} \\
\quad \quad \text{If } \text{FPSCR}_{OE} = 1 \text{ then goto Enabled Exponent Overflow} \\
\quad \text{End} \\
\text{If } (\text{FRB})_{1:11} > 896 \text{ and } (\text{FRB})_{1:11} < 1151 \text{ then goto Normal Operand} \\
\text{If } (\text{FRB})_{1:63} = 0 \text{ then goto Zero Operand} \\
\text{If } (\text{FRB})_{1:11} = 2047 \text{ then} \\
\quad \text{Do} \\
\quad \quad \text{If } (\text{FRB})_{12:63} = 0 \text{ then goto Infinity Operand} \\
\quad \quad \text{If } (\text{FRB})_{12} = 1 \text{ then goto QNaN Operand} \\
\quad \quad \text{If } (\text{FRB})_{12} = 0 \text{ and } (\text{FRB})_{13:63} > 0 \text{ then goto SNaN Operand} \\
\quad \text{End} \\
\text{Disabled Exponent Underflow:} \\
\quad \text{sign} \leftarrow (\text{FRB})_0 \\
\quad \text{If } (\text{FRB})_{1:11} = 0 \text{ then} \\
\quad \quad \text{Do} \\
\quad \quad \quad \text{exp} \leftarrow -1022 \\
\quad \quad \quad \text{frac}_{0:52} \leftarrow 0b0 && (\text{FRB})_{12:63} \\
\quad \quad \text{End} \\
\quad \text{If } (\text{FRB})_{1:11} > 0 \text{ then} \\
\quad \quad \text{Do} \\
\quad \quad \quad \text{exp} \leftarrow (\text{FRB})_{1:11} - 1023 \\
\quad \quad \quad \text{frac}_{0:52} \leftarrow 0b1 && (\text{FRB})_{12:63} \\
\quad \text{End} \\
\quad \text{Denormalize operand:} \\
\quad \quad \text{G} \ || \ R \ || \ X \leftarrow 0b000 \\
\quad \text{Do while } \text{exp} < -126 \\
\quad \quad \quad \text{exp} \leftarrow \text{exp} + 1 \\
\quad \quad \quad \text{frac}_{0:52} \ || \ G \ || \ R \ || \ X \leftarrow 0b0 \ || \ frac_{0:52} \ || \ G \ || \ (R \ || X) \\
\quad \text{End} \\
\quad \text{FPSCR}_{UX} \leftarrow (\text{frac}_{24:52} \ || \ G \ || \ R \ || X) > 0 \\
\quad \text{Round Single}(	ext{sign,exp,frac}_{0:52},G,R,X) \\
\quad \text{FPSCR}_{XX} \leftarrow \text{FPSCR}_{XX} \ || \ \text{FPSCR}_{FI} \\
\quad \text{If } \text{frac}_{0:52} = 0 \text{ then} \\
\quad \quad \text{Do} \\
\end{align*}
\]
FRT\textsubscript{0} \leftarrow \text{sign}
FRT\textsubscript{1,63} \leftarrow 0
\text{If sign} = 0 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "+ zero"
\text{If sign} = 1 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "- zero"
\text{End}
\text{If frac}_0:52 > 0 \text{ then}
\text{Do}
\text{If frac}_0 = 1 \text{ then}
\text{Do}
\text{If sign} = 0 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "+ normal number"
\text{If sign} = 1 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "- normal number"
\text{End}
\text{If frac}_0 = 0 \text{ then}
\text{Do}
\text{If sign} = 0 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "+ denormalized number"
\text{If sign} = 1 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "- denormalized number"
\text{End}
\text{Normalize operand:}
\text{Do while frac}_0 = 0
\text{exp} \leftarrow \text{exp}-1
\text{frac}_0:52 \leftarrow \text{frac}_1:52 \|	ext{0b0}
\text{End}
FRT\textsubscript{0} \leftarrow \text{sign}
FRT\textsubscript{1,11} \leftarrow \text{exp} + 1023
FRT\textsubscript{12,63} \leftarrow \text{frac}_1:52
\text{End}
\text{Done}

\textbf{Enabled Exponent Underflow:}
\text{FPSCR}_{\text{UX}} \leftarrow 1
\text{sign} \leftarrow (\text{FRB})_0
\text{If (FRB)}_{1,11} = 0 \text{ then}
\text{Do}
\text{exp} \leftarrow -1022
\text{frac}_0:52 \leftarrow \text{0b0} \|	ext{(FRB)}_{12,63}
\text{End}
\text{If (FRB)}_{1,11} > 0 \text{ then}
\text{Do}
\text{exp} \leftarrow (\text{FRB})_{1,11} - 1023
\text{frac}_0:52 \leftarrow \text{0b1} \|	ext{(FRB)}_{12,63}
\text{End}
\text{Normalize operand:}
\text{Do while frac}_0 = 0
\text{exp} \leftarrow \text{exp} - 1
\text{frac}_0:52 \leftarrow \text{frac}_1:52 \|	ext{0b0}
\text{End}
\text{Round Single(sign,exp,frac}_0:52,0,0,0,0)
\text{FPSCR}_{\text{XX}} \leftarrow \text{FPSCR}_{\text{XX}} \|	ext{FPSCR}_{\text{Fl}}
\text{exp} \leftarrow \text{exp} + 192
FRT\textsubscript{0} \leftarrow \text{sign}
FRT\textsubscript{1,11} \leftarrow \text{exp} + 1023
FRT\textsubscript{12,63} \leftarrow \text{frac}_1:52
\text{If sign} = 0 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "+ normal number"
\text{If sign} = 1 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "- normal number"
\text{Done}

\textbf{Disabled Exponent Overflow:}
\text{FPSCR}_{\text{OX}} \leftarrow 1
\text{If FPSCR}_{\text{RN}} = \text{0b00} \text{ then} \quad /^{*} \text{Round to Nearest} */
\text{Do}
\text{If (FRB)}_0 = 0 \text{ then } \text{FRT} \leftarrow \text{0xFF0_0000_0000_0000}_0
\text{If (FRB)}_0 = 1 \text{ then } \text{FRT} \leftarrow \text{0xFFF0_0000_0000_0000}_0
\text{If (FRB)}_0 = 0 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "+ infinity"
\text{If (FRB)}_0 = 1 \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "- infinity"
\text{End}
If \( \text{FPSCR}_{RN} = 0b01 \) then /* Round toward Zero */
   Do
      If \((\text{FRB})_0 = 0 \) then \( \text{FRT} \leftarrow 0x47EF_{FFFF}\_E000\_0000 \)
      If \((\text{FRB})_0 = 1 \) then \( \text{FRT} \leftarrow 0xC7EF_{FFFF}\_E000\_0000 \)
      If \((\text{FRB})_0 = 0 \) then \( \text{FPSCR}_{FPRF} \leftarrow +\text{ normal number} \)
      If \((\text{FRB})_0 = 1 \) then \( \text{FPSCR}_{FPRF} \leftarrow -\text{ normal number} \)
   End
If \( \text{FPSCR}_{RN} = 0b10 \) then /* Round toward +Infinity */
   Do
      If \((\text{FRB})_0 = 0 \) then \( \text{FRT} \leftarrow 0x7FF0_{0000}\_0000\_0000 \)
      If \((\text{FRB})_0 = 1 \) then \( \text{FRT} \leftarrow 0xC7EF_{FFFF}\_E000\_0000 \)
      If \((\text{FRB})_0 = 0 \) then \( \text{FPSCR}_{FPRF} \leftarrow +\text{ infinity} \)
      If \((\text{FRB})_0 = 1 \) then \( \text{FPSCR}_{FPRF} \leftarrow -\text{ normal number} \)
   End
If \( \text{FPSCR}_{RN} = 0b11 \) then /* Round toward -Infinity */
   Do
      If \((\text{FRB})_0 = 0 \) then \( \text{FRT} \leftarrow 0x47EF_{FFFF}\_E000\_0000 \)
      If \((\text{FRB})_0 = 1 \) then \( \text{FRT} \leftarrow 0xFFF0_{0000}\_0000\_0000 \)
      If \((\text{FRB})_0 = 0 \) then \( \text{FPSCR}_{FPRF} \leftarrow +\text{ normal number} \)
      If \((\text{FRB})_0 = 1 \) then \( \text{FPSCR}_{FPRF} \leftarrow -\text{ infinity} \)
   End
\( \text{FPSCR}_{FR} \leftarrow \text{undefined} \)
\( \text{FPSCR}_{FI} \leftarrow 1 \)
\( \text{FPSCR}_{XX} \leftarrow 1 \)
Done

Enabled Exponent Overflow:

\[
\begin{align*}
\text{sign} & \leftarrow (\text{FRB})_0 \\
\text{exp} & \leftarrow (\text{FRB})_{1:11} - 1023 \\
\text{frac}_{0:52} & \leftarrow 0b1 \|$ (\text{FRB})_{12:63} \\
\text{Round Single}(\text{sign}, \text{exp}, \text{frac}_{0:52}, 0, 0, 0) \\
\text{FPSCR}_{XX} & \leftarrow \text{FPSCR}_{XX} \|$ \text{FPSCR}_{FI}
\end{align*}
\]

Enabled Overflow:

\[
\begin{align*}
\text{FPSCR}_{OX} & \leftarrow 1 \\
\text{exp} & \leftarrow \text{exp} - 192 \\
\text{FRT}_0 & \leftarrow \text{sign} \\
\text{FRT}_{1:11} & \leftarrow \text{exp} + 1023 \\
\text{FRT}_{12:63} & \leftarrow \text{frac}_{1:52} \\
\text{If} \text{ sign } = 0 \text{ then } \text{FPSCR}_{FPRF} \leftarrow +\text{ normal number} \\
\text{If} \text{ sign } = 1 \text{ then } \text{FPSCR}_{FPRF} \leftarrow -\text{ normal number} \\
\text{Done}
\end{align*}
\]

Zero Operand:

\[
\begin{align*}
\text{FRT} & \leftarrow (\text{FRB}) \\
\text{If} \text{ (FRB)}_0 = 0 \text{ then } \text{FPSCR}_{FPRF} \leftarrow +\text{ zero} \\
\text{If} \text{ (FRB)}_0 = 1 \text{ then } \text{FPSCR}_{FPRF} \leftarrow -\text{ zero} \\
\text{FPSCR}_{FR} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

Infinity Operand:

\[
\begin{align*}
\text{FRT} & \leftarrow (\text{FRB}) \\
\text{If} \text{ (FRB)}_0 = 0 \text{ then } \text{FPSCR}_{FPRF} \leftarrow +\text{ infinity} \\
\text{If} \text{ (FRB)}_0 = 1 \text{ then } \text{FPSCR}_{FPRF} \leftarrow -\text{ infinity} \\
\text{FPSCR}_{FR} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

QNaN Operand:

\[
\begin{align*}
\text{FRT} & \leftarrow (\text{FRB})_{0:34} \|$ 2^{9} \\
\text{FPSCR}_{FPRF} & \leftarrow \text{"QNaN"} \\
\text{FPSCR}_{FR} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]
SNaN Operand:

 FPSCR_{VXSNAN} \leftarrow 1
 If FPSCR_{VE} = 0 then
 Do
 \begin{align*}
   & FRT_{0:11} \leftarrow (FRB)_{0:11} \\
   & FRT_{12} \leftarrow 1 \\
   & FRT_{13:63} \leftarrow (FRB)_{13:34} || 290 \\
   & FPSCR_{FPRF} \leftarrow "QNaN"
 \end{align*}
 End
 FPSCR_{FRFI} \leftarrow 0b00
 Done

Normal Operand:

 sign \leftarrow (FRB)_0
 exp \leftarrow (FRB)_{1:11} - 1023
 frac_{0:52} \leftarrow 0b1 || (FRB)_{12:63}
 Round Single(sign,exp,frac_{0:52},0,0,0)
 FPSCR_{XX} \leftarrow FPSCR_{XX} | FPSCR_{FI}
 If exp > 127 and FPSCR_{OE} = 0 then go to Disabled Exponent Overflow
 If exp > 127 and FPSCR_{OE} = 1 then go to Enabled Overflow
 FRT_0 \leftarrow sign
 FRT_{1:11} \leftarrow exp + 1023
 FRT_{12:63} \leftarrow frac_{1:52}
 If sign = 0 then FPSCR_{FPRF} \leftarrow "+ normal number"
 If sign = 1 then FPSCR_{FPRF} \leftarrow "- normal number"
 Done

Round Single(sign,exp,frac_{0:52},G,R,X):

 inc \leftarrow 0
 lsb \leftarrow frac_{23}
 gbit \leftarrow frac_{24}
 rbit \leftarrow frac_{25}
 xbit \leftarrow (frac_{26:52}||G||R||X) = 0
 If FPSCR_{RN} = 0b00 then /* Round to Nearest */
 Do /* comparisons ignore u bits */
 \begin{align*}
   & If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc \leftarrow 1 \\
   & If sign || lsb || gbit || rbit || xbit = 0bu011u then inc \leftarrow 1 \\
   & If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc \leftarrow 1
 \end{align*}
 End
 If FPSCR_{RN} = 0b10 then /* Round toward + Infinity */
 Do /* comparisons ignore u bits */
 \begin{align*}
   & If sign || lsb || gbit || rbit || xbit = 0b01uu then inc \leftarrow 1 \\
   & If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc \leftarrow 1 \\
   & If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc \leftarrow 1
 \end{align*}
 End
 If FPSCR_{RN} = 0b11 then /* Round toward - Infinity */
 Do /* comparisons ignore u bits */
 \begin{align*}
   & If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc \leftarrow 1 \\
   & If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc \leftarrow 1 \\
   & If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc \leftarrow 1
 \end{align*}
 End
 frac_{0:23} \leftarrow frac_{0:23} + inc
 If carry_out = 1 then
 Do
 \begin{align*}
   & frac_{0:23} \leftarrow 0b1 || frac_{0:22} \\
   & exp \leftarrow exp + 1
 \end{align*}
 End
 frac_{24:52} \leftarrow 290
 FPSCR_{FR} \leftarrow inc
 FPSCR_{FI} \leftarrow gbit || rbit || xbit
 Return
A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the Floating Convert To Integer instructions.

if Floating Convert To Integer Word then do
    round_mode ← FPSCR屁
    tgt_precision ← “32-bit signed integer”
end
if Floating Convert To Integer Word Unsigned then do
    round_mode ← FPSCR屁
    tgt_precision ← “32-bit unsigned integer”
end
if Floating Convert To Integer Word with round toward Zero then do
    round_mode ← 0b01
    tgt_precision ← “32-bit signed integer”
end
if Floating Convert To Integer Word Unsigned with round toward Zero then do
    round_mode ← 0b01
    tgt_precision ← “32-bit unsigned integer”
end
if Floating Convert To Integer Doubleword then do
    round_mode ← FPSCR屁
    tgt_precision ← “64-bit signed integer”
end
if Floating Convert To Integer Doubleword Unsigned then do
    round_mode ← FPSCR屁
    tgt_precision ← “64-bit unsigned integer”
end
if Floating Convert To Integer Doubleword with round toward Zero then do
    round_mode ← 0b01
    tgt_precision ← “64-bit signed integer”
end
if Floating Convert To Integer Doubleword Unsigned with round toward Zero then do
    round_mode ← 0b01
    tgt_precision ← “64-bit unsigned integer”
end

sign ← (FRB)_0
if (FRB)_1:11 = 2047 and (FRB)_12:63 = 0 then goto Infinity Operand
if (FRB)_1:11 = 2047 and (FRB)_12 = 0 then goto SNaN Operand
if (FRB)_1:11 = 2047 and (FRB)_12 = 1 then goto QNaN Operand
if (FRB)_1:11 > 1086 then goto Large Operand
if (FRB)_1:11 > 0 then exp ← (FRB)_1:11 - 1023 /* exp - bias */
if (FRB)_1:11 = 0 then exp ← -1022
if (FRB)_1:11 > 0 then frac0:64 ← 0b01 || (FRB)_12:63 || 110 /* normal */
if (FRB)_1:11 = 0 then frac0:64 ← 0b00 || (FRB)_12:63 || 110 /* denormal */
gbit || rbit || xbit ← 0b000
do i=1,63-exp /* do the loop 0 times if exp = 63 */
    frac0:64 || gbit || rbit || xbit ← 0b00 || frac0:64 || gbit || (rbit | xbit)
end
Round Integer( sign, frac0:64, gbit, rbit, xbit, round_mode )
if sign = 1 then frac0:64 ← ¬frac0:64 + 1 /* needed leading 0 for -2^{64}<(FRB)<-2^{63} */
if tgt_precision = “32-bit signed integer” and frac0:64 > 2^{31}-1 then
goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 > 2^{63}-1 then
goto Large Operand
if tgt_precision = “32-bit signed integer” and frac0:64 < -2^{31} then
goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 < -2^{63} then
goto Large Operand
if tgt_precision = “32-bit unsigned integer” & frac0:64 > 2^{32}-1 then
goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 > 2^{64}-1 then
goto Large Operand
if tgt_precision = “32-bit unsigned integer” & frac0:64 < 0 then
goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 < 0 then
goto Large Operand

FPSCRXX ← FPSCRXX | FPSCR_FI

if tgt_precision = “32-bit signed integer” then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “64-bit signed integer” then FRT ← frac1:64
if tgt_precision = “64-bit unsigned integer” then FRT ← frac1:64

Round Integer( sign, frac0:64, gbit, rbit, xbit, round_mode ):

inc ← 0
if round_mode = 0b00 then do /* Round to Nearest */
    if sign || frac64 || gbit || rbit || xbit = 0bU11UU then inc ← 1
    if sign || frac64 || gbit || rbit || xbit = 0bU01UU then inc ← 1
end
if round_mode = 0b10 then do /* Round toward +Infinity */
    if sign || frac64 || gbit || rbit || xbit = 0b0U1UU then inc ← 1
    if sign || frac64 || gbit || rbit || xbit = 0b0UU1U then inc ← 1
end
if round_mode = 0b11 then do /* Round toward -Infinity */
    if sign || frac64 || gbit || rbit || xbit = 0b1U1UU then inc ← 1
    if sign || frac64 || gbit || rbit || xbit = 0b1UU1U then inc ← 1
end
frac0:64 ← frac0:64 + inc
FPSCR_FPR ← inc
FPSCR_FI ← gbit | rbit | xbit
return

Infinity Operand:
FPSCR_FPR ← 0b0
FPSCR_FI ← 0b0
FPSCR_FXCVT ← 0b1
if FPSCR_VE = 0 then do
    if tgt_precision = “32-bit signed integer” then do
        if sign=0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
        if sign=1 then FRT ← 0xUUUU_UUUU_8000_0000
    end
    else if tgt_precision = “32-bit unsigned integer” then do
        if sign=0 then FRT ← 0xUUUU_UUUU_0000_0000
        if sign=1 then FRT ← 0xUUUU_UUUU_0000_0000
    end
    else if tgt_precision = “64-bit signed integer” then do
        if sign=0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
        if sign=1 then FRT ← 0x8000_0000_0000_0000
    end
end

else if tgt_precision = "64-bit unsigned integer" then do
  if sign=0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
  if sign=1 then FRT ← 0x0000_0000_0000_0000
end
FPSCR_FPRF ← 0bUUUUUU
end
done

SNaN Operand:

FPSCR_FPR ← 0b0
FPSCR_FRI ← 0b0
FPSCR_XSNAN ← 0b1
FPSCR_XCVI ← 0b1
if FPSCR_VE = 0 then do
  if tgt_precision = "32-bit signed integer" then FRT ← 0xUUUU_UUUU_8000_0000
  if tgt_precision = "64-bit signed integer" then FRT ← 0x8000_0000_0000_0000
  if tgt_precision = "32-bit unsigned integer" then FRT ← 0xUUUU_UUUU_0000_0000
  if tgt_precision = "64-bit unsigned integer" then FRT ← 0x0000_0000_0000_0000
  FPSCR_FPRF ← 0bUUUUUU
end
done

QNaN Operand:

FPSCR_FPR ← 0b0
FPSCR_FRI ← 0b0
FPSCR_XCVI ← 0b1
if FPSCR_VE = 0 then do
  if tgt_precision = "32-bit signed integer" then FRT ← 0xUUUU_UUUU_8000_0000
  if tgt_precision = "64-bit signed integer" then FRT ← 0x8000_0000_0000_0000
  if tgt_precision = "32-bit unsigned integer" then FRT ← 0xUUUU_UUUU_0000_0000
  if tgt_precision = "64-bit unsigned integer" then FRT ← 0x0000_0000_0000_0000
  FPSCR_FPRF ← 0bUUUUUU
end
done

Large Operand:

FPSCR_FPR ← 0b0
FPSCR_FRI ← 0b0
FPSCR_XCVI ← 0b1
if FPSCR_VE = 0 then do
  if tgt_precision = "32-bit signed integer" then do
    if sign = 0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
    if sign = 1 then FRT ← 0xUUUU_UUUU_8000_0000
  end
  else if tgt_precision = "64-bit signed integer" then do
    if sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
    if sign = 1 then FRT ← 0x8000_0000_0000_0000
  end
  else if tgt_precision = "32-bit unsigned integer" then do
    if sign = 0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
    if sign = 1 then FRT ← 0xUUUU_UUUU_0000_0000
  end
  else if tgt_precision = "64-bit unsigned integer" then do
    if sign = 0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
    if sign = 1 then FRT ← 0x0000_0000_0000_0000
  end
  FPSCR_FPRF ← 0bUUUUUU
end
done
A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer* instructions.

```plaintext
if Floating Convert From Integer Doubleword then do
    tgt_precision ← "double-precision"
    sign ← (FRB)_0
    exp ← 63
    frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Single then do
    tgt_precision ← "single-precision"
    sign ← (FRB)_0
    exp ← 63
    frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Unsigned then do
    tgt_precision ← "double-precision"
    sign ← 0
    exp ← 63
    frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Unsigned Single then do
    tgt_precision ← "single-precision"
    sign ← 0
    exp ← 63
    frac0:63 ← (FRB)
end
if frac0:63 = 0 then go to Zero Operand
if sign = 1 then frac0:63 ← ¬frac0:63 + 1
/* do the loop 0 times if (FRB) = max negative 64-bit integer or */
/*                     if (FRB) = max unsigned 64-bit integer    */
do while frac0 = 0
    frac0:63 ← frac1:63 || 0b0
    exp ← exp - 1
end
Round Float( sign, exp, frac0:63, RN )
if sign = 0 then FPSCR_FPRF ← "+ normal number"
if sign = 1 then FPSCR_FPRF ← "- normal number"
FRT0 ← sign
FRT1:11 ← exp + 1023 /* exp + bias */
FRT12:63 ← frac1:52
done

Zero Operand:
FPSCR_FFR ← 0b00
FPSCR_FFI ← 0b00
FPSCR_FPRF ← "+ zero"
FRT ← 0x0000_0000_0000_0000_0000_0000
done

Round Float( sign, exp, frac0:63, round_mode ):
    inc ← 0
    if tgt_precision = "single-precision" then do
        lsb ← frac23
        gbit ← frac24
        rbit ← frac25
        xbit ← frac26:63 > 0
    end
else do /* tgt_precision = "double-precision" */
```
lsb ← frac_{52}
gbit ← frac_{53}
rbit ← frac_{54}
xbit ← frac_{55:63} > 0

end

if round_mode = 0b00 then do /* Round to Nearest */
    if sign || lsb || gbit || rbit || xbit = 0bU11UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0bU011U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end

if round_mode = 0b10 then do /* Round toward + Infinity */
    if sign || lsb || gbit || rbit || xbit = 0b0U1UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b0UU1U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end

if round_mode = 0b11 then do /* Round toward - Infinity */
    if sign || lsb || gbit || rbit || xbit = 0b1U1UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b1UU1U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end

if tgt_precision = “single-precision” then
    frac_{0:23} ← frac_{0:23} + inc
else /* tgt_precision = “double-precision” */
    frac_{0:52} ← frac_{0:52} + inc

if carry_out = 1 then exp ← exp + 1

FPSCR_{FR} ← inc
FPSCR_{FI} ← gbit | rbit | xbit
FPSCR_{XX} ← FPSCR_{XX} | FPSCR_{FI}
return
A.4 Floating-Point Round to Integer Model

The following describes algorithmically the operation of the Floating Round To Integer instructions.

If (FRB)_{1:11} = 2047 and (FRB)_{12:63} = 0, then goto Infinity Operand
If (FRB)_{1:11} = 2047 and (FRB)_{12} = 0, then goto SNaN Operand
If (FRB)_{1:11} = 2047 and (FRB)_{12} = 1, then goto QNaN Operand
If (FRB)_{1:63} = 0 then goto Zero Operand /* exp < 0; |value| < 1*/
If (FRB)_{1:11} < 1023 then goto Small Operand /* exp < 0; |value| < 1*/
If (FRB)_{1:11} > 1074 then goto Large Operand /* exp > 51; integral value */

sign ← (FRB)_{0}
exp ← (FRB)_{1:11} - 1023 /* exp - bias */
frac_{0:52} ← 0b1 || (FRB)_{12:63}
gbit || rbit || xbit ← 0b000

Do i = 1, 52 - exp
   frac_{0:52} || gbit || rbit || xbit ← 0b0 || frac_{0:52} || gbit || (rbit || xbit)
End

Round Integer (sign, frac_{0:52}, gbit, rbit, xbit)

Do i = 2, 52 - exp
   frac_{0:52} ← frac_{1:52} || 0b0
End

If frac_{0} = 1, then exp ← exp + 1
Else frac_{0:52} ← frac_{1:52} || 0b0

FRT_{0} ← sign
FRT_{1:11} ← exp + 1023
FRT_{12:63} ← frac_{1:52}

If (FRT)_{0} = 0 then FPSCR_{FPRF} ← "+ normal number"
Else FPSCR_{FPRF} ← "- normal number"
FPSCR_{FR F} ← 0b00
Done

Round Integer(sign, frac_{0:52}, gbit, rbit, xbit):

inc ← 0
If inst = Floating Round to Integer Nearest then /* ties away from zero */
   Do /* comparisons ignore u bits */
      If sign || frac_{52} || gbit || rbit || xbit = 0buu1uu then inc ← 1
   End
If inst = Floating Round to Integer Plus then
   Do /* comparisons ignore u bits */
      If sign || frac_{52} || gbit || rbit || xbit = 0b0u1uu then inc ← 1
      If sign || frac_{52} || gbit || rbit || xbit = 0b0uu1u then inc ← 1
      If sign || frac_{52} || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
   End
If inst = Floating Round to Integer Minus then
   Do /* comparisons ignore u bits */
      If sign || frac_{52} || gbit || rbit || xbit = 0b1u1uu then inc ← 1
      If sign || frac_{52} || gbit || rbit || xbit = 0b1uu1u then inc ← 1
      If sign || frac_{52} || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
   End
frac_{0:52} ← frac_{0:52} + inc
Return
Infinity Operand:

\[
\begin{align*}
\text{FRT} & \leftarrow (\text{FRB}) \\
\text{If } (\text{FRB})_0 = 0 & \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "+\text{ infinity}" \\
\text{If } (\text{FRB})_0 = 1 & \text{ then } \text{FPSCR}_{\text{FPRF}} \leftarrow "-\text{ infinity}" \\
\text{FPSCR}_{\text{FR FI}} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

SNaN Operand:

\[
\begin{align*}
\text{FPSCR}_{\text{VXSNAN}} & \leftarrow 1 \\
\text{If } \text{FPSCR}_{\text{VE}} = 0 & \text{ then} \\
\text{Do} & \\
\text{FRT} & \leftarrow (\text{FRB}) \\
\text{FRT}_{12} & \leftarrow 1 \\
\text{FPSCR}_{\text{FPRF}} & \leftarrow "\text{QNaN}" \\
\text{End} \\
\text{FPSCR}_{\text{FR FI}} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

QNaN Operand:

\[
\begin{align*}
\text{FRT} & \leftarrow (\text{FRB}) \\
\text{FPSCR}_{\text{FPRF}} & \leftarrow "\text{QNaN}" \\
\text{FPSCR}_{\text{FR FI}} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

Zero Operand:

\[
\begin{align*}
\text{If } (\text{FRB})_0 = 0 & \text{ then} \\
\text{Do} & \\
\text{FRT} & \leftarrow 0x0000\_0000\_0000\_0000 \\
\text{FPSCR}_{\text{FPRF}} & \leftarrow "+\text{ zero}" \\
\text{End} \\
\text{Else} & \\
\text{Do} & \\
\text{FRT} & \leftarrow 0x8000\_0000\_0000\_0000 \\
\text{FPSCR}_{\text{FPRF}} & \leftarrow "-\text{ zero}" \\
\text{End} \\
\text{FPSCR}_{\text{FR FI}} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

Small Operand:

If inst = Floating Round to Integer Nearest and 
\((\text{FRB})_{11} \leq 1022\) then goto Zero Operand
If inst = Floating Round to Integer Toward Zero 
then goto Zero Operand
If inst = Floating Round to Integer Plus and 
\((\text{FRB})_0 = 1\) then goto Zero Operand
If inst = Floating Round to Integer Minus and 
\((\text{FRB})_0 = 0\) then goto Zero Operand
If \((\text{FRB})_0 = 0\) then 
\[
\begin{align*}
\text{Do} & \\
\text{FRT} & \leftarrow 0x3FF0\_0000\_0000\_0000 \\
\text{ /* value = 1.0 */ } \\
\text{FPSCR}_{\text{FPRF}} & \leftarrow "+\text{ normal number}" \\
\text{End} \\
\text{Else} & \\
\text{Do} & \\
\text{FRT} & \leftarrow 0xBFF0\_0000\_0000\_0000 \\
\text{ /* value = -1.0 */ } \\
\text{FPSCR}_{\text{FPRF}} & \leftarrow "-\text{ normal number}" \\
\text{End} \\
\text{FPSCR}_{\text{FR FI}} & \leftarrow 0b00 \\
\text{Done}
\end{align*}
\]

Large Operand:

\[
\begin{align*}
\text{FRT} & \leftarrow (\text{FRB})
\end{align*}
\]
Appendix B. Densely Packed Decimal

The trailing significand field of the decimal floating-point data format is encoded using Densely Packed Decimal (DPD). DPD encoding is a compression technique which supports the representation of decimal integers of arbitrary length. Translation operates on three Binary Coded Decimal (BCD) digits at a time compressing the 12 bits into 10 bits with an algorithm that can be applied or reversed using simple Boolean operations. In the following examples, a 3-digit BCD number is represented as (abcd)(efgh)(ijklm), a 10-bit DPD number is represented as (pqrstuv)(wxyz), and the Boolean operations, & (AND), | (OR), and ~ (NOT) are used.

B.1 BCD-to-DPD Translation

The translation from a 3-digit BCD number to a 10-bit DPD can be performed through the following Boolean operations.

\[
\begin{align*}
p &= (f \& a \& i \& \neg e) \mid (j \& a \& \neg i) \mid (b \& \neg a) \\
q &= (g \& a \& i \& \neg e) \mid (k \& a \& \neg i) \mid (c \& \neg a) \\
r &= d \\
s &= (j \& \neg a \& e \& \neg i) \mid (f \& \neg i \& \neg e) \\
t &= (k \& \neg a \& e \& \neg i) \mid (g \& \neg i \& \neg e) \\
u &= h \\
v &= a \mid e \mid i \\
w &= (\neg e \& j \& \neg i) \mid (e \& i) \mid a \\
x &= (\neg a \& k \& \neg i) \mid (a \& i) \mid e \\
y &= m
\end{align*}
\]

Alternatively, the following table can be used to perform the translation. The most significant bit of the three BCD digits (left column) is used to select a specific 10-bit encoding (right column) of the DPD.

<table>
<thead>
<tr>
<th>sei</th>
<th>pqr stu v wxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>bcd fgh 0 jkm</td>
</tr>
<tr>
<td>001</td>
<td>bcd fgh 1 00m</td>
</tr>
<tr>
<td>010</td>
<td>bcd jkh 1 01m</td>
</tr>
<tr>
<td>011</td>
<td>bcd 10h 1 11m</td>
</tr>
<tr>
<td>100</td>
<td>jkd fgh 1 10m</td>
</tr>
<tr>
<td>101</td>
<td>fgd 01h 1 11m</td>
</tr>
<tr>
<td>110</td>
<td>jkd 00h 1 11m</td>
</tr>
<tr>
<td>111</td>
<td>00d 11h 1 11m</td>
</tr>
</tbody>
</table>

The full translation of a 3-digit BCD number (000 - 999) to a 10-bit DPD is shown in Table 123 on page 699.

B.2 DPD-to-BCD Translation

The translation from a 10-bit DPD to a 3-digit BCD number can be performed through the following Boolean operations.

\[
\begin{align*}
a &= (\neg s \& v \& w) \mid (t \& v \& w \& s) \mid (v \& w \& \neg x) \\
b &= (p \& s \& x \& \neg t) \mid (p \& \neg w) \mid (p \& \neg v) \\
c &= (q \& s \& x \& \neg t) \mid (q \& \neg w) \mid (q \& \neg v) \\
d &= r \\
e &= (v \& \neg w \& x) \mid (s \& v \& w \& x) \mid (\neg t \& v \& x \& w) \\
f &= (p \& t \& v \& w \& x \& \neg s) \mid (s \& \neg x \& v) \mid (s \& \neg v) \\
g &= (q \& t \& w \& v \& x \& \neg s) \mid (t \& \neg x \& v) \mid (t \& \neg v) \\
h &= u \\
i &= (t \& v \& w \& x) \mid (s \& v \& w \& x) \mid (v \& \neg w \& \neg x) \\
j &= (p \& \neg s \& \neg t \& w \& v) \mid (s \& v \& \neg w \& x) \mid (p \& w \& \neg x \& v) \mid (w \& \neg v) \\
k &= (q \& \neg s \& \neg t \& v \& w) \mid (t \& v \& \neg w \& x) \mid (q \& v \& w \& \neg x) \mid (x \& \neg v) \\
m &= y
\end{align*}
\]

Alternatively, the following table can be used to perform the translation. A combination of five bits in the DPD encoding (leftmost column) are used to specify a translation to the 3-digit BCD encoding. Dashes (-) in the table are don’t cares, and can be either one or zero.
The full translation of the 10-bit DPD to a 3-digit BCD number is shown in Table 124 on page 700. The 10-bit DPD index is produced by concatenating the 6-bit value shown in the left column with the 4-bit index along the top row, both represented in hexadecimal. The values in parentheses are non-preferred translations and are explained further in the following section.

### B.3 Preferred DPD encoding

Translating from a 3-digit BCD number (1000 numbers) to a 10-bit DPD encoding (1024 combinations) leaves 24 redundant translations. The 24 redundant combinations are evenly assigned to eight BCD numbers and are shown in the following table, with the non-preferred encoding in parentheses. The preferred encoding is produced by translating a 3-digit BCD number with the translation table or Boolean operations shown in Section B.1. The redundant DPD encodings are all valid and will be correctly translated to their respective BCD value through the mechanisms provided in Section B.2. For decimal floating-point operations all DPD encodings are recognized as source operands.
<table>
<thead>
<tr>
<th>00</th>
<th>000</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
<th>005</th>
<th>006</th>
<th>007</th>
<th>008</th>
<th>009</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>010</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td>015</td>
<td>016</td>
<td>017</td>
<td>018</td>
<td>019</td>
</tr>
<tr>
<td>02</td>
<td>020</td>
<td>021</td>
<td>022</td>
<td>023</td>
<td>024</td>
<td>025</td>
<td>026</td>
<td>027</td>
<td>028</td>
<td>029</td>
</tr>
<tr>
<td>03</td>
<td>030</td>
<td>031</td>
<td>032</td>
<td>033</td>
<td>034</td>
<td>035</td>
<td>036</td>
<td>037</td>
<td>038</td>
<td>039</td>
</tr>
<tr>
<td>04</td>
<td>040</td>
<td>041</td>
<td>042</td>
<td>043</td>
<td>044</td>
<td>045</td>
<td>046</td>
<td>047</td>
<td>048</td>
<td>049</td>
</tr>
<tr>
<td>05</td>
<td>050</td>
<td>051</td>
<td>052</td>
<td>053</td>
<td>054</td>
<td>055</td>
<td>056</td>
<td>057</td>
<td>058</td>
<td>059</td>
</tr>
<tr>
<td>06</td>
<td>060</td>
<td>061</td>
<td>062</td>
<td>063</td>
<td>064</td>
<td>065</td>
<td>066</td>
<td>067</td>
<td>068</td>
<td>069</td>
</tr>
<tr>
<td>07</td>
<td>070</td>
<td>071</td>
<td>072</td>
<td>073</td>
<td>074</td>
<td>075</td>
<td>076</td>
<td>077</td>
<td>078</td>
<td>079</td>
</tr>
<tr>
<td>08</td>
<td>080</td>
<td>081</td>
<td>082</td>
<td>083</td>
<td>084</td>
<td>085</td>
<td>086</td>
<td>087</td>
<td>088</td>
<td>089</td>
</tr>
<tr>
<td>09</td>
<td>090</td>
<td>091</td>
<td>092</td>
<td>093</td>
<td>094</td>
<td>095</td>
<td>096</td>
<td>097</td>
<td>098</td>
<td>099</td>
</tr>
</tbody>
</table>

Table 123: BCD-to-DPD translation

<table>
<thead>
<tr>
<th>00</th>
<th>000</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
<th>005</th>
<th>006</th>
<th>007</th>
<th>008</th>
<th>009</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>010</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td>015</td>
<td>016</td>
<td>017</td>
<td>018</td>
<td>019</td>
</tr>
<tr>
<td>02</td>
<td>020</td>
<td>021</td>
<td>022</td>
<td>023</td>
<td>024</td>
<td>025</td>
<td>026</td>
<td>027</td>
<td>028</td>
<td>029</td>
</tr>
<tr>
<td>03</td>
<td>030</td>
<td>031</td>
<td>032</td>
<td>033</td>
<td>034</td>
<td>035</td>
<td>036</td>
<td>037</td>
<td>038</td>
<td>039</td>
</tr>
<tr>
<td>04</td>
<td>040</td>
<td>041</td>
<td>042</td>
<td>043</td>
<td>044</td>
<td>045</td>
<td>046</td>
<td>047</td>
<td>048</td>
<td>049</td>
</tr>
<tr>
<td>05</td>
<td>050</td>
<td>051</td>
<td>052</td>
<td>053</td>
<td>054</td>
<td>055</td>
<td>056</td>
<td>057</td>
<td>058</td>
<td>059</td>
</tr>
<tr>
<td>06</td>
<td>060</td>
<td>061</td>
<td>062</td>
<td>063</td>
<td>064</td>
<td>065</td>
<td>066</td>
<td>067</td>
<td>068</td>
<td>069</td>
</tr>
<tr>
<td>07</td>
<td>070</td>
<td>071</td>
<td>072</td>
<td>073</td>
<td>074</td>
<td>075</td>
<td>076</td>
<td>077</td>
<td>078</td>
<td>079</td>
</tr>
<tr>
<td>08</td>
<td>080</td>
<td>081</td>
<td>082</td>
<td>083</td>
<td>084</td>
<td>085</td>
<td>086</td>
<td>087</td>
<td>088</td>
<td>089</td>
</tr>
<tr>
<td>09</td>
<td>090</td>
<td>091</td>
<td>092</td>
<td>093</td>
<td>094</td>
<td>095</td>
<td>096</td>
<td>097</td>
<td>098</td>
<td>099</td>
</tr>
</tbody>
</table>

Appendix B. Densely Packed Decimal 699
## Table 124: DPD-to-BCD Translation

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>001</td>
<td>002</td>
<td>003</td>
<td>004</td>
<td>005</td>
<td>006</td>
<td>007</td>
<td>008</td>
<td>009</td>
<td>010</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>015</td>
<td>016</td>
<td>017</td>
<td>018</td>
<td>019</td>
<td>020</td>
<td>021</td>
<td>022</td>
<td>023</td>
<td>024</td>
<td>025</td>
<td>026</td>
<td>027</td>
<td>028</td>
<td>029</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>030</td>
<td>031</td>
<td>032</td>
<td>033</td>
<td>034</td>
<td>035</td>
<td>036</td>
<td>037</td>
<td>038</td>
<td>039</td>
<td>040</td>
<td>041</td>
<td>042</td>
<td>043</td>
<td>044</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>045</td>
<td>046</td>
<td>047</td>
<td>048</td>
<td>049</td>
<td>050</td>
<td>051</td>
<td>052</td>
<td>053</td>
<td>054</td>
<td>055</td>
<td>056</td>
<td>057</td>
<td>058</td>
<td>059</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>060</td>
<td>061</td>
<td>062</td>
<td>063</td>
<td>064</td>
<td>065</td>
<td>066</td>
<td>067</td>
<td>068</td>
<td>069</td>
<td>070</td>
<td>071</td>
<td>072</td>
<td>073</td>
<td>074</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>075</td>
<td>076</td>
<td>077</td>
<td>078</td>
<td>079</td>
<td>080</td>
<td>081</td>
<td>082</td>
<td>083</td>
<td>084</td>
<td>085</td>
<td>086</td>
<td>087</td>
<td>088</td>
<td>089</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>090</td>
<td>091</td>
<td>092</td>
<td>093</td>
<td>094</td>
<td>095</td>
<td>096</td>
<td>097</td>
<td>098</td>
<td>099</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>0A</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>0B</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
<td>190</td>
<td>191</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
<td>207</td>
<td>208</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td>210</td>
<td>211</td>
<td>212</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>216</td>
<td>217</td>
<td>218</td>
<td>219</td>
<td>220</td>
<td>221</td>
<td>222</td>
<td>223</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td>225</td>
<td>226</td>
<td>227</td>
<td>228</td>
<td>229</td>
<td>230</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>234</td>
<td>235</td>
<td>236</td>
<td>237</td>
<td>238</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>240</td>
<td>241</td>
<td>242</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>246</td>
<td>247</td>
<td>248</td>
<td>249</td>
<td>250</td>
<td>251</td>
<td>252</td>
<td>253</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>255</td>
<td>256</td>
<td>257</td>
<td>258</td>
<td>259</td>
<td>260</td>
<td>261</td>
<td>262</td>
<td>263</td>
<td>264</td>
<td>265</td>
<td>266</td>
<td>267</td>
<td>268</td>
<td>269</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>270</td>
<td>271</td>
<td>272</td>
<td>273</td>
<td>274</td>
<td>275</td>
<td>276</td>
<td>277</td>
<td>278</td>
<td>279</td>
<td>280</td>
<td>281</td>
<td>282</td>
<td>283</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>285</td>
<td>286</td>
<td>287</td>
<td>288</td>
<td>289</td>
<td>290</td>
<td>291</td>
<td>292</td>
<td>293</td>
<td>294</td>
<td>295</td>
<td>296</td>
<td>297</td>
<td>298</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>300</td>
<td>301</td>
<td>302</td>
<td>303</td>
<td>304</td>
<td>305</td>
<td>306</td>
<td>307</td>
<td>308</td>
<td>309</td>
<td>310</td>
<td>311</td>
<td>312</td>
<td>313</td>
<td>314</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>315</td>
<td>316</td>
<td>317</td>
<td>318</td>
<td>319</td>
<td>320</td>
<td>321</td>
<td>322</td>
<td>323</td>
<td>324</td>
<td>325</td>
<td>326</td>
<td>327</td>
<td>328</td>
<td>329</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>330</td>
<td>331</td>
<td>332</td>
<td>333</td>
<td>334</td>
<td>335</td>
<td>336</td>
<td>337</td>
<td>338</td>
<td>339</td>
<td>340</td>
<td>341</td>
<td>342</td>
<td>343</td>
<td>344</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>345</td>
<td>346</td>
<td>347</td>
<td>348</td>
<td>349</td>
<td>350</td>
<td>351</td>
<td>352</td>
<td>353</td>
<td>354</td>
<td>355</td>
<td>356</td>
<td>357</td>
<td>358</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>360</td>
<td>361</td>
<td>362</td>
<td>363</td>
<td>364</td>
<td>365</td>
<td>366</td>
<td>367</td>
<td>368</td>
<td>369</td>
<td>370</td>
<td>371</td>
<td>372</td>
<td>373</td>
<td>374</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>375</td>
<td>376</td>
<td>377</td>
<td>378</td>
<td>379</td>
<td>380</td>
<td>381</td>
<td>382</td>
<td>383</td>
<td>384</td>
<td>385</td>
<td>386</td>
<td>387</td>
<td>388</td>
<td>389</td>
<td></td>
</tr>
</tbody>
</table>

**Power ISA™ - Book I**
Appendix C. Vector RTL Functions [Category: Vector]

ConvertSPtoSXW saturate (X, Y)

    sign  = X0
    exp0:7 = X1:8
    frac0:30 = X9:31 || 0b0000_0000
    if((exp=255) && (frac!=0)) then return(0x0000_0000) // NaN operand
    if((exp=255) && (frac==0)) then do // infinity operand
        VSCRSAT = 1
        if((exp+Y-127)>30) then do // large operand
            VSCRSAT = 1
            if((sign==1) ? 0x0000_0000 : 0x7FFF_FFFF
        if((exp+Y-127)<0) then return(0x0000_0000) // -1.0 < value < 1.0 (value rounds to 0)
        significand0:31 = 0b1
        do i=1 to 31-(exp+Y-127)
            significand = significand >> ui 1
        return( (sign==0) ? significand : (¬significand + 1) )

ConvertSPtoUXW saturate (X, Y)

    sign  = X0
    exp0:7 = X1:8
    frac0:30 = X9:31 || 0b0000_0000
    if((exp=255) && (frac!=0)) then return(0x0000_0000) // NaN operand
    if((exp=255) && (frac==0)) then do // infinity operand
        VSCRSAT = 1
        return( (sign==1) ? 0x0000_0000 : 0xFFFF_FFFF
        if((exp+Y-127)>31) then do // large operand
            VSCRSAT = 1
            if((exp+Y-127)<0) then return(0x0000_0000) // -1.0 < value < 1.0 (value rounds to 0)
            if( sign==1 ) then do // negative operand
                fracsat = 1
                return(0x0000_0000)
        significand0:31 = 0b1 || frac
        do i=1 to 31-(exp+Y-127)
            significand = significand >> ui 1
        return( significand )

ConvertSXWtoSP (X)

    sign  = X0
    exp0:7 = 32 + 127
    frac0:32 = X0 || X0:31
    if( frac=0 ) return(0x0000_0000) // Zero operand
    if( sign=1 ) then frac = ¬frac + 1
    do while( frac0=0 )
        frac = frac << 1
        exp = exp - 1
        lsb = frac0 || 0b3
        gbit = frac24 || 0b3
        xbit = frac25:32!=0
        inc = ( lsb && gbit ) || ( gbit && xbit )
        frac0:32 = frac0:32 || inc
        if( carry_out=1 ) exp = exp + 1
    return( sign || exp || frac0:32 )
ConvertUXWtoSP( X )

exp0:7 = 31 + 127
frac0:31 = X0:31

if( frac==0 ) return( 0x0000_0000 ) // Zero Operand

do while( frac==0 )
    frac = frac << 1
    exp = exp - 1

lsb = frac23
gbit = frac24
xbit = frac25:31!=0

inc = ( lsb && gbit ) | ( gbit && xbit )
frac0:23 = frac0:23 + inc

if( carry_out==1 ) exp = exp + 1

return( 0b0 || exp || frac1:23 )
Appendix D. Embedded Floating-Point RTL Functions

[Category: SPE.Embedded Float Scalar Double]
[Category: SPE.Embedded Float Scalar Single]
[Category: SPE.Embedded Float Vector]

D.1 Common Functions

// Check if 32-bit fp value is a NaN or Infinity
Isa32NaNorInfinity(fp)
return (fpexp = 255)

Isa32NaN(fp)
return ((fpexp = 255) & (fpfrac ≠ 0))

// Check if 32-bit fp value is denormalized
Isa32Denorm(fp)
return ((fpexp = 0) & (fpfrac ≠ 0))

// Check if 64-bit fp value is a NaN or Infinity
Isa64NaNorInfinity(fp)
return (fpexp = 2047)

Isa64NaN(fp)
return ((fpexp = 2047) & (fpfrac ≠ 0))

// Check if 32-bit fp value is denormalized
Isa64Denorm(fp)
return ((fpexp = 0) & (fpfrac ≠ 0))

// Signal an error in the SPEFSCR
SignalFPError(upper_lower, bits)
if (upper_lower = HI) then
  bits ← bits << 15
SPEFSCR ← SPEFSCR | bits
if (upper_lower = HI) then
  bits ← (FG | FX)
SPEFSCR ← SPEFSCR & ~bits

// Round a 32-bit fp result
Round32(fp, guard, sticky)
FP32format fp;
if (SPEFSCRFINXE = 0) then
  if (SPEFSCRFRMC = 0b00) then // nearest
    if (guard) then
      if (sticky) then
        fpfrac ← fpfrac + 1
      if (fpexp = 0) then
        fp ← fp ≤ 254 then // overflow
          fp ← fp ≤ sign || 0b11111111 || 231
        else
          fpexp ← fpexp + 1
          fpfrac ← v1:23
        else if ((SPEFSCRFRMC & 0b10) = 0b10) then
          // infinity modes
          // implementation dependent
          return fp
    // Round a 64-bit fp result
Round64(fp, guard, sticky)
FP32format fp;
if (SPEFSCRFINXE = 0) then
  if (SPEFSCRFRMC = 0b00) then // nearest
    if (guard) then
      if (sticky) then
        fpfrac ← fpfrac + 1
      if (fpexp = 0) then
        fp ← fp ≤ 2046 then // overflow
          fp ← fp ≤ sign || 0b1111111111 || 521
        else
          fpexp ← fpexp + 1
          fpfrac ← v1:52
        else if ((SPEFSCRFRMC & 0b10) = 0b10) then
          // infinity modes
          // implementation dependent
          return fp
D.2 Convert from Single-Precision Embedded Floating-Point to Integer Word with Saturation

// Convert 32-bit Floating-Point to 32-bit integer
// or fractional
// signed = S (signed) or U (unsigned)
// upper_lower = HI (high word) or LO (low word)
// round = RND (round) or ZER (truncate)
// fractional = F (fractional) or I (integer)

CnvtfP32ToI32Sat(fp, signed, upper_lower, round, fractional)

FP32Format fp;
if (Isa32NaNorInfinity(fp)) then
  SignalFPError(upper_lower, FINV)
if (Isa32NaN(fp)) then
  return 0x00000000 // all NaNs
if (signed = S) then
  if (fpsign = 1) then
    return 0x80000000
  else
    return 0x7fffffff
else
  if (fpsign = 1) then
    return 0x00000000
  else
    return 0xffffffff
if (Isa32Denorm(fp)) then
  SignalFPError(upper_lower, FINV)
  return 0x00000000 // regardless of sign
if ((signed = U) & (fpsign = 1)) then
  SignalFPError(upper_lower, FOVF) // overflow
  return 0x00000000
if ((fpexp = 0) & (fpfrac = 0)) then
  return 0x00000000 // all zero values
if (fractional = I) then // convert to integer
  max_exp ← 158
  shift ← 158 - fpexp
  if (signed = S) then
    if ((fpexp ≠ 158) | (fpfrac ≠ 0) | (fpsign ≠ 1)) then
      max_exp ← max_exp - 1
  else // fractional conversion
    max_exp ← 126
    shift ← 126 - fpexp
  if (signed = S) then
    shift ← shift + 1
  if (fpexp > max_exp) then
    SignalFPError(upper_lower, FOVF) // overflow
  if (signed = S) then
    if (fpsign = 1) then
      return 0x80000000
    else
      return 0x7fffffff
  else
    return 0xffffffff
result ← 0b1 || fpfrac || 0b00000000 // add U bit
guard ← 0
sticky ← 0
for (n ← 0; n < shift; n ← n + 1) do
  sticky ← sticky | guard
  guard ← result & 0x00000001
  result ← result > 1
// Report sticky and guard bits
if (upper_lower = HI) then
  SPEFSCR_FPD ← guard
  SPEFSCR_FPX ← sticky
else
  SPEFSCR_FPD ← guard
  SPEFSCR_FPX ← sticky
if (guard | sticky) then
  SPEFSCR_FINXS ← 1
// Round the integer result
if ((round = RND) & (SPEFSCR_FINXR = 0)) then
  if (SPEFSCR_FPD = 0b00) then // nearest
    if (stick) then (result & 0x00000001) then
      result ← result + 1
    else if ((SPEFSCR_FPD & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
  if (signed = S) then
    if (fpsign = 1) then
      result ← ¬result + 1
return result
D.3 Convert from Double-Precision Embedded Floating-Point to Integer Word with Saturation

// Convert 64-bit Floating-Point to 32-bit integer
// or fractional
// signed = S (signed) or U (unsigned)
// round = RND (round) or ZER (truncate)
// fractional = F (fractional) or I (integer)

Cnvtp64ToI32Sat(fp, signed, round, fractional)
FP64format fp;

if (Isa64NaNorInfinity(fp)) then
  SignalFPError(L0, FINV)
if (Isa64NaN(fp)) then
  return 0x00000000 // all NaNs
if (signed = S) then
  if (fpsign = 1) then
    return 0x80000000
  else
    return 0x7fffffff
else
  if (fpsign = 1) then
    return 0x00000000
  else
    return 0xffffffff
if (Isa64Denorm(fp)) then
  SignalFPError(L0, FINV)
  return 0x00000000 // regardless of sign
if ((signed = U) & (fpsgn = 1)) then
  SignalFPError(L0, FOVF) // overflow
  return 0x00000000
if ((fpexp = 0) & (fpfrac = 0)) then
  return 0x00000000 // all zero values
if (fractional = I) then // convert to integer
  max_exp ← 1054
  shift ← 1054 - fpexp
  if (signed ← S) then
    if ((fpexp < 1054) & (fpfrac ≠ 0) & (fpsign = 1)) then
      max_exp ← max_exp - 1
  else // fractional conversion
    max_exp ← 1022
    shift ← 1022 - fpexp
  if (signed = S) then
    shift ← shift + 1
if (fpexp > max_exp) then
  SignalFPError(L0, FOVF) // overflow
  if (signed = S) then
    if (fpsgn = 1) then
      return 0x80000000
    else
      return 0xffffffff
  else
    return 0xffffffff
result ← 0b1 || fpfrac[0:30] // add U to frac
guard ← result & 0x00000001
result ← result > 1
// Report sticky and guard bits
SPEFSCRPU ← guard
SPEFSCRFX ← sticky
if (guard | sticky) then
  SPEFSCRFINXS ← 1
// Round the result
if ((round = RND) & (SPEFSCRFINX = 0)) then
  if (SPEFSCRFRMC = 0b00) then // nearest
    if (guard) then
      if (sticky | (result & 0x00000001)) then
        result ← result + 1
    else if ((SPEFSCRFINX & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
  if (signed = S) then
    if (fpsgn = 1) then
      result ← ¬result + 1
  return result
D.4 Convert from Double-Precision Embedded Floating-Point to Integer Doubleword with Saturation

// Convert 64-bit Floating-Point to 64-bit integer
// signed = S (signed) or U (unsigned)
// round = RND (round) or ZER (truncate)

CnvtFP64ToI64Sat(fp, signed, round)
FP64Format fp;
if (Isa64NaNorInfinity(fp)) then
    SignalFPErr(L0, FINV)
    if (Isa64NaN(fp)) then
        return 0x00000000_00000000 // all NaNs
        if (signed = S) then
            if (fpsign = 1) then
                return 0x80000000_00000000
            else
                return 0x7fffffff_ffffffff
        else
            if (fpsign = 1) then
                return 0x00000000_00000000
            else
                return 0xffffffff_ffffffff
    if (Isa64Denorm(fp)) then
        SignalFPErr(L0, FINV)
        return 0x00000000_00000000
    if ((signed = U) & (fpsign = 1)) then
        SignalFPErr(L0, FOVF) // overflow
        return 0x00000000_00000000
    if ((fpexp = 0) & (fpfrac = 0)) then
        return 0x00000000_00000000 // all zero values
    max_exp ← 1086
    shift ← 1086 - fpexp
    if (signed = S) then
        if ((fpexp ≤ 1086) & (fpfrac = 0) & (fpsign = 1)) then
            max_exp ← max_exp - 1
        if (fpexp > max_exp) then
            SignalFPErr(L0, POVF) // overflow
            if (signed = S) then
                if (fpsign = 1) then
                    return 0x80000000_00000000
                else
                    return 0x7fffffff_ffffffff
            else
                return 0xffffffff_ffffffff
    result ← 0b1 || fpfrac || 0b00000000000 //add U bit
    guard ← 0
    sticky ← 0
    for (n ← 0; n < shift; n ← n + 1) do
        sticky ← sticky | guard
        guard ← result & 0x00000000_00000001
        result ← result > 1
    // Report sticky and guard bits
    SPEFSCRPG ← guard
    SPEFSCRPS ← sticky
    if (guard | sticky) then
        SPEFSCRPS ← 1
    // Round the result
    if ((round = RND) & (SPEFSCRPS = 0)) then
        if (SPEFSCRPOE = 0b00) then // nearest
            if (guard) then
                result ← result + 1
            else if ((SPEFSCRPOE & 0b10) = 0b10) then
                // infinity modes
                // implementation dependent
        if (signed = S) then
            if (fpsign = 1) then
                result ← ¬result + 1
            return result
D.5 Convert to Single-Precision Embedded Floating-Point from Integer Word

// Convert from 32-bit integer or fractional to 32-bit Floating-Point
// signed = S (signed) or U (unsigned)
// round = RND (round) or ZER (truncate)
// fractional = F (fractional) or I (integer)

CnvtI32ToFp32(v, signed, upper_lower, fractional)
FP32format result;
resultsign ← 0
if (v = 0) then
result ← 0
if (upper_lower = HI) then
SPEFSCRFGH ← 0
SPEFSCRFXH ← 0
else
SPEFSCRFG ← 0
SPEFSCRFX ← 0
else
if (signed = S) then
if (v0 = 1) then
v ← ¬v + 1
resultsign ← 1
if (fractional = F) then // frac bit align
maxexp ← 127
if (signed = U) then
maxexp ← maxexp - 1
else
maxexp ← 158 // integer bit alignment
sc ← 0
while (v0 = 0)
v ← v << 1
sc ← sc + 1
v0 ← 0 // clear U bit
resultexp ← maxexp - sc
guard ← v24
sticky ← (v25:31 ≠ 0)
// Report sticky and guard bits
if (upper_lower = HI) then
SPEFSCRFGH ← guard
SPEFSCRFXH ← sticky
else
SPEFSCRFG ← guard
SPEFSCRFX ← sticky
if (guard | sticky) then
SPEFSCRFINXS ← 1
// Round the result
resultfrac ← v1:23
result ← Round32(result, guard, sticky)
return result

D.6 Convert to Double-Precision Embedded Floating-Point from Integer Word

// Convert from integer or fractional to 64 bit Floating-Point
// signed = S (signed) or U (unsigned)
// fractional = F (fractional) or I (integer)

CnvtI32ToFp64(v, signed, fractional)
FP64format result;
resultsign ← 0
if (v = 0) then
result ← 0
SPEFSCRFGV0 ← 0
SPEFSCRFXV0 ← 0
else
if (signed = S) then
if (v0 = 1) then
v ← ¬v + 1
resultsign ← 1
if (fractional = F) then // frac bit align
maxexp ← 1023
if (signed = U) then
maxexp ← maxexp - 1
else
maxexp ← 1054 // integer bit align
sc ← 0
while (v0 = 0)
v ← v << 1
sc ← sc + 1
v0 ← 0 // clear U bit
resultexp ← maxexp - sc
// Report sticky and guard bits
SPEFSCRFGV0 ← 0
SPEFSCRFXV0 ← 0
resultfrac ← v1:31 || 210
return result
D.7 Convert to Double-Precision Embedded Floating-Point from Integer Doubleword

// Convert from 64-bit integer to 64-bit floating-point
// signed = S (signed) or U (unsigned)

CnvtI64ToFp64(v, signed)
FP64format result;
result.sign ← 0
if (v = 0) then
    result ← 0
    SPEFSFPG ← 0
    SPEFSFPGX ← 0
else
    if (signed = S) then
        if (v0 = 1) then
            v ← ¬v + 1
            result.sign ← 1
        maxexp ← 1054
        sc ← 0
        while (v0 = 0)
            v ← v << 1
            sc ← sc + 1
        v0 ← 0 // clear U bit
        result.exp ← maxexp - sc
        guard ← v53
        sticky ← (v54:63 ≠ 0)

    // Report sticky and guard bits
    SPEFSFPG ← guard
    SPEFSFPGX ← sticky
    if (guard | sticky) then
        SPEFSFPGX ← 1

    // Round the result
    result.frac ← v1:52
    result ← Round64(result, guard, sticky)

return result
Appendix E. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of Branch Conditional, Compare, Trap, Rotate and Shift, and certain other instructions.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

E.1 Symbols

The following symbols are defined for use in instructions (basic or extended mnemonics) that specify a Condition Register field or a Condition Register bit. The first five (lt, ..., un) identify a bit number within a CR field. The remainder (cr0, ..., cr7) identify a CR field. An expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol and 32 can be used to identify a CR bit.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>0</td>
<td>Less than</td>
</tr>
<tr>
<td>gt</td>
<td>1</td>
<td>Greater than</td>
</tr>
<tr>
<td>eq</td>
<td>2</td>
<td>Equal</td>
</tr>
<tr>
<td>so</td>
<td>3</td>
<td>Summary overflow</td>
</tr>
<tr>
<td>un</td>
<td>3</td>
<td>Unordered (after floating-point comparison)</td>
</tr>
<tr>
<td>cr0</td>
<td>0</td>
<td>CR Field 0</td>
</tr>
<tr>
<td>cr1</td>
<td>1</td>
<td>CR Field 1</td>
</tr>
<tr>
<td>cr2</td>
<td>2</td>
<td>CR Field 2</td>
</tr>
<tr>
<td>cr3</td>
<td>3</td>
<td>CR Field 3</td>
</tr>
<tr>
<td>cr4</td>
<td>4</td>
<td>CR Field 4</td>
</tr>
<tr>
<td>cr5</td>
<td>5</td>
<td>CR Field 5</td>
</tr>
<tr>
<td>cr6</td>
<td>6</td>
<td>CR Field 6</td>
</tr>
<tr>
<td>cr7</td>
<td>7</td>
<td>CR Field 7</td>
</tr>
</tbody>
</table>

The extended mnemonics in Sections E.2.2 and E.3 require identification of a CR bit: if one of the CR field symbols is used, it must be multiplied by 4 and added to a bit-number-within-CR-field (value in the range 0-3, explicit or symbolic) and 32. The extended mnemonics in Sections E.2.3 and E.5 require identification of a CR field: if one of the CR field symbols is used, it must not be multiplied by 4 or added to 32. (For the extended mnemonics in Section E.2.3, the bit number within the CR field is part of the extended mnemonic. The programmer identifies the CR field, and the Assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)
E.2 Branch Mnemonics

The mnemonics discussed in this section are variations of the Branch Conditional instructions.

Note: bclr, bclrl, bcctr, and bcctrl each serve as both a basic and an extended mnemonic. The Assembler will recognize a bclr, bclrl, bcctr, or bcctrl mnemonic with three operands as the basic form, and a bclr, bclrl, bcctr, or bcctrl mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00. Similarly, for all the extended mnemonics described in Sections E.2.2 - E.2.4 that devolve to any of these four basic mnemonics the BH operand can either be coded or omitted. If it is omitted it is assumed to be 0b00.

E.2.1 BO and BI Fields

The 5-bit BO and BI fields control whether the branch is taken. Providing an extended mnemonic for every possible combination of these fields would be neither useful nor practical. The mnemonics described in Sections E.2.2 - E.2.4 include the most useful cases. Other cases can be coded using a basic Branch Conditional mnemonic (bc[l][a], bclr[l], bcctr[l]) with the appropriate operands.

E.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 125 that tests a Condition Register bit specify the corresponding bit as the first operand. The symbols defined in Section E.1 can be used in this operand.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics b, ba, bl, and bla should be used.

<table>
<thead>
<tr>
<th>Branch Semantics</th>
<th>LR not Set</th>
<th>LR Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bc Relative</td>
<td>bca Absolute</td>
</tr>
<tr>
<td>Branch unconditionally</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Branch if CRBI=1</td>
<td>bt</td>
<td>bta</td>
</tr>
<tr>
<td>Branch if CRBI=0</td>
<td>bf</td>
<td>bfa</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero</td>
<td>bdnz</td>
<td>bdnza</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero and CRBI=1</td>
<td>bdnzt</td>
<td>bdnzta</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR nonzero and CRBI=0</td>
<td>bdnzf</td>
<td>bdnzfa</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero</td>
<td>bdz</td>
<td>bdza</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero and CRBI=1</td>
<td>bdzt</td>
<td>bdzta</td>
</tr>
<tr>
<td>Decrement CTR, branch if CTR zero and CRBI=0</td>
<td>bdzf</td>
<td>bdzfa</td>
</tr>
</tbody>
</table>

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).
   
   bdnz target (equivalent to: bc 16,0,target)

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is "equal".
   
   bdnzt eq,target (equivalent to: bc 8,2,target)

3. Same as (2), but "equal" condition is in CR5.
   
   bdnzt 4×cr5+eq,target (equivalent to: bc 8,22,target)
4. Branch if bit 59 of CR is 0.
   \texttt{bf} 27,target  \quad (equivalent to:  \texttt{bc} 4,27,target)

5. Same as (4), but set the Link Register. This is a form of conditional “call”.
   \texttt{bfl} 27,target  \quad (equivalent to:  \texttt{bcl} 4,27,target)

### E.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table 126, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table 126 specify the CR field as an optional first operand. One of the CR field symbols defined in Section E.1 can be used for this operand. If the CR field being tested is CR Field 0, this operand need not be specified unless the resulting basic mnemonic is \texttt{bclr} or \texttt{bcctr} and the BH operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Less than</td>
</tr>
<tr>
<td>le</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>eq</td>
<td>Equal</td>
</tr>
<tr>
<td>ge</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
</tr>
<tr>
<td>nl</td>
<td>Not less than</td>
</tr>
<tr>
<td>ne</td>
<td>Not equal</td>
</tr>
<tr>
<td>ng</td>
<td>Not greater than</td>
</tr>
<tr>
<td>so</td>
<td>Summary overflow</td>
</tr>
<tr>
<td>ns</td>
<td>Not summary overflow</td>
</tr>
<tr>
<td>un</td>
<td>Unordered (after floating-point comparison)</td>
</tr>
<tr>
<td>nu</td>
<td>Not unordered (after floating-point comparison)</td>
</tr>
</tbody>
</table>

These codes are reflected in the mnemonics shown in Table 126.

<table>
<thead>
<tr>
<th>Branch Semantics</th>
<th>LR not Set</th>
<th>LR Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bc Relative</td>
<td>bca Absolute</td>
</tr>
<tr>
<td>Branch if less than</td>
<td>bit</td>
<td>bila</td>
</tr>
<tr>
<td>Branch if less than or equal</td>
<td>ble</td>
<td>blea</td>
</tr>
<tr>
<td>Branch if equal</td>
<td>beq</td>
<td>beqa</td>
</tr>
<tr>
<td>Branch if greater than or equal</td>
<td>bge</td>
<td>bgea</td>
</tr>
<tr>
<td>Branch if greater than</td>
<td>bgt</td>
<td>bgta</td>
</tr>
<tr>
<td>Branch if not less than</td>
<td>bnl</td>
<td>bnlta</td>
</tr>
<tr>
<td>Branch if not equal</td>
<td>bne</td>
<td>bneqa</td>
</tr>
<tr>
<td>Branch if not greater than</td>
<td>bng</td>
<td>bngta</td>
</tr>
<tr>
<td>Branch if summary overflow</td>
<td>bso</td>
<td>bsoa</td>
</tr>
<tr>
<td>Branch if not summary overflow</td>
<td>bns</td>
<td>bnsa</td>
</tr>
<tr>
<td>Branch if unordered</td>
<td>bun</td>
<td>bunra</td>
</tr>
<tr>
<td>Branch if not unordered</td>
<td>bnu</td>
<td>bnuia</td>
</tr>
</tbody>
</table>

### Examples

1. Branch if CR0 reflects condition “not equal”.
   \texttt{bne} target  \quad (equivalent to:  \texttt{bc} 4,2,target)

2. Same as (1), but condition is in CR3.
bne cr3,target (equivalent to: bc 4,14,target)

3. Branch to an absolute target if CR4 specifies “greater than”, setting the Link Register. This is a form of conditional “call”.
   bgta cr4,target (equivalent to: bcla 12,17,target)

4. Same as (3), but target address is in the Count Register.
   bgtctrl cr4 (equivalent to: bcctrl 12,17,0)

**E.2.4 Branch Prediction**

Software can use the “at” bits of *Branch Conditional* instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

+ Predict branch to be taken (at=0b11)
- Predict branch not to be taken (at=0b10)

Such a suffix can be added to any *Branch Conditional* mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

**Examples**

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.
   blt+ target

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.
   bltlr-
E.3 Condition Register Logical Mnemonics

The Condition Register Logical instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

Table 127: Condition Register logical mnemonics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Register set</td>
<td>crset bx</td>
<td>creqv bx,bx,bx</td>
</tr>
<tr>
<td>Condition Register clear</td>
<td>crclr bx</td>
<td>crxor bx,bx,bx</td>
</tr>
<tr>
<td>Condition Register move</td>
<td>crmove bx,by</td>
<td>cror bx,by,by</td>
</tr>
<tr>
<td>Condition Register not</td>
<td>crnot bx,by</td>
<td>cmor bx,by,by</td>
</tr>
</tbody>
</table>

The symbols defined in Section E.1 can be used to identify the Condition Register bits.

Examples
1. Set CR bit 57.
   
   crset 25 (equivalent to: creqv 25,25,25)
2. Clear the SO bit of CR0.
   
   crclr so (equivalent to: crxor 3,3,3)
3. Same as (2), but SO bit to be cleared is in CR3.
   
   crclr 4x cr3+so (equivalent to: crxor 15,15,15)
4. Invert the EQ bit.
   
   crnot eq,eq (equivalent to: cmor 2,2,2)
5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.
   
   crnot 4x cr5+eq,4x cr4+eq (equivalent to: cmor 22,18,18)

E.4 Subtract Mnemonics

E.4.1 Subtract Immediate

Although there is no “Subtract Immediate” instruction, its effect can be achieved by using an Add Immediate instruction with the immediate operand negated. Extended mnemonics are provided that include this negation, making the intent of the computation clearer.

   subi Rx,Ry,value (equivalent to: addi Rx,Ry,-value)
   subis Rx,Ry,value (equivalent to: addis Rx,Ry,-value)
   subic Rx,Ry,value (equivalent to: addic Rx,Ry,-value)
   subic Rx,Ry,value (equivalent to: addic Rx,Ry,-value)

E.4.2 Subtract

The Subtract From instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more “normal” order, in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final “o” and/or “c” to cause the OE and/or Rc bit to be set in the underlying instruction.

   sub Rx,Ry,Rz (equivalent to: subf Rx,Rz,Ry)
   subc Rx,Ry,Rz (equivalent to: subfc Rx,Rz,Ry)
E.5 Compare Mnemonics

The L field in the fixed-point Compare instructions controls whether the operands are treated as 64-bit quantities or as 32-bit quantities. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The BF field can be omitted if the result of the comparison is to be placed into CR Field 0. Otherwise the target CR field must be specified as the first operand. One of the CR field symbols defined in Section E.1 can be used for this operand.

Note: The basic Compare mnemonics of Power ISA are the same as those of POWER, but the POWER instructions have three operands while the Power ISA instructions have four. The Assembler will recognize a basic Compare mnemonic with three operands as the POWER form, and will generate the instruction with L=0. (Thus the Assembler must require that the BF field, which normally can be omitted when CR Field 0 is the target, be specified explicitly if L is.)

E.5.1 Doubleword Comparisons

Table 128: Doubleword compare mnemonics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare doubleword immediate</td>
<td>cmpdi bf,ra,si</td>
<td>cmpi bf,1,ra,si</td>
</tr>
<tr>
<td>Compare doubleword</td>
<td>cmpd bf,ra,rb</td>
<td>cmp bf,1,ra,rb</td>
</tr>
<tr>
<td>Compare logical doubleword immediate</td>
<td>cmpldi bf,ra,ui</td>
<td>cmpli bf,1,ra,ui</td>
</tr>
<tr>
<td>Compare logical doubleword</td>
<td>cmpld bf,ra,rb</td>
<td>cmpl bf,1,ra,rb</td>
</tr>
</tbody>
</table>

Examples

1. Compare register Rx and immediate value 100 as unsigned 64-bit integers and place result into CR0.
   
   ```
   cmpidi Rx,100
   ```
   (equivalent to:
   ```
   cmpli 0,1,Rx,100
   ```)

2. Same as (1), but place result into CR4.
   
   ```
   cmpidi cr4,Rx,100
   ```
   (equivalent to:
   ```
   cmpli 4,1,Rx,100
   ```)

3. Compare registers Rx and Ry as signed 64-bit integers and place result into CR0.
   
   ```
   cmpd Rx,Ry
   ```
   (equivalent to:
   ```
   cmp 0,1,Rx,Ry
   ```)

E.5.2 Word Comparisons

Table 129: Word compare mnemonics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare word immediate</td>
<td>cmpwi bf,ra,si</td>
<td>cmpi bf,0,ra,si</td>
</tr>
<tr>
<td>Compare word</td>
<td>cmpw bf,ra,rb</td>
<td>cmp bf,0,ra,rb</td>
</tr>
<tr>
<td>Compare logical word immediate</td>
<td>cmplwi bf,ra,ui</td>
<td>cmpli bf,0,ra,ui</td>
</tr>
<tr>
<td>Compare logical word</td>
<td>cmplw bf,ra,rb</td>
<td>cmp bf,0,ra,rb</td>
</tr>
</tbody>
</table>

Examples

1. Compare bits 32:63 of register Rx and immediate value 100 as signed 32-bit integers and place result into CR0.
   
   ```
   cmpwi Rx,100
   ```
   (equivalent to:
   ```
   cmpli 0,0,Rx,100
   ```)

2. Same as (1), but place result into CR4.
   
   ```
   cmpwi cr4,Rx,100
   ```
   (equivalent to:
   ```
   cmpli 4,0,Rx,100
   ```)

3. Compare bits 32:63 of registers Rx and Ry as unsigned 32-bit integers and place result into CR0.
   
   ```
   cmplw Rx,Ry
   ```
   (equivalent to:
   ```
   cmp 0,0,Rx,Ry
   ```)
E.6 Trap Mnemonics

The mnemonics defined in Table 130 are variations of the Trap instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of trap conditions.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>TO encoding</th>
<th>&lt;</th>
<th>&gt;</th>
<th>=</th>
<th>&lt;u</th>
<th>&gt;u</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Less than</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>le</td>
<td>Less than or equal</td>
<td>20</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eq</td>
<td>Equal</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ge</td>
<td>Greater than or equal</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nl</td>
<td>Not less than</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ne</td>
<td>Not equal</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ng</td>
<td>Not greater than</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>llt</td>
<td>Logically less than</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lle</td>
<td>Logically less than or equal</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lge</td>
<td>Logically greater than or equal</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lgt</td>
<td>Logically greater than</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lnl</td>
<td>Logically not less than</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lng</td>
<td>Logically not greater than</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>Unconditionally with parameters</td>
<td>31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(none)</td>
<td>Unconditional</td>
<td>31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

These codes are reflected in the mnemonics shown in Table 130.

<table>
<thead>
<tr>
<th>Trap Semantics</th>
<th>64-bit Comparison</th>
<th>32-bit Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tdi Immediate</td>
<td>td Register</td>
</tr>
<tr>
<td>Trap unconditionally</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Trap unconditionally with parameters</td>
<td>tduim</td>
<td>tdum</td>
</tr>
<tr>
<td>Trap if less than</td>
<td>tdlit</td>
<td>tdl</td>
</tr>
<tr>
<td>Trap if less than or equal</td>
<td>tdliei</td>
<td>tdle</td>
</tr>
<tr>
<td>Trap if equal</td>
<td>tdeqi</td>
<td>tdeq</td>
</tr>
<tr>
<td>Trap if greater than or equal</td>
<td>tdgei</td>
<td>tdge</td>
</tr>
<tr>
<td>Trap if greater than</td>
<td>tdgiti</td>
<td>tdgt</td>
</tr>
<tr>
<td>Trap if not less than</td>
<td>tdnli</td>
<td>tdnl</td>
</tr>
<tr>
<td>Trap if not equal</td>
<td>tdnei</td>
<td>tdne</td>
</tr>
<tr>
<td>Trap if not greater than</td>
<td>tdnji</td>
<td>tdnj</td>
</tr>
<tr>
<td>Trap if logically less than</td>
<td>tdltli</td>
<td>tdltl</td>
</tr>
<tr>
<td>Trap if logically less than or equal</td>
<td>tdlliei</td>
<td>tdllle</td>
</tr>
<tr>
<td>Trap if logically greater than or equal</td>
<td>tdlgiei</td>
<td>tdlg</td>
</tr>
<tr>
<td>Trap if logically greater than</td>
<td>tdlgiti</td>
<td>tdlg</td>
</tr>
<tr>
<td>Trap if logically not less than</td>
<td>tdnl</td>
<td>tdnl</td>
</tr>
<tr>
<td>Trap if logically not greater than</td>
<td>tdngi</td>
<td>tdng</td>
</tr>
</tbody>
</table>
Examples

1. Trap if register Rx is not 0.
   \[\text{tnei } Rx,0 \quad \text{(equivalent to: } tdi \quad 24,Rx,0)\]

2. Same as (1), but comparison is to register Ry.
   \[\text{tne } Rx,Ry \quad \text{(equivalent to: } td \quad 24,Rx,Ry)\]

3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.
   \[\text{twlgti } Rx,0x7FF \quad \text{(equivalent to: } twi \quad 1,Rx,0x7FF)\]

4. Trap unconditionally.
   \[\text{trap} \quad \text{(equivalent to: } tw \quad 31,0,0)\]

5. Trap unconditionally with immediate parameters Rx and Ry
   \[\text{tdu } Rx,Ry \quad \text{(equivalent to: } td \quad 31,Rx,Ry)\]

E.7 Integer Select Mnemonics

The mnemonics defined in Table 131, “Integer Select mnemonics,” on page 716 are variations of the Integer Select instructions, with the most useful values of BC represented in the mnemonic rather than specified as a numeric operand.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>Less than</td>
</tr>
<tr>
<td>eq</td>
<td>Equal</td>
</tr>
<tr>
<td>gt</td>
<td>Greater than</td>
</tr>
</tbody>
</table>

These codes are reflected in the mnemonics shown in Table 131.

<table>
<thead>
<tr>
<th>Table 131: Integer Select mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Select semantics</strong></td>
</tr>
<tr>
<td>Integer Select if less than</td>
</tr>
<tr>
<td>Integer Select if equal</td>
</tr>
<tr>
<td>Integer Select if greater than</td>
</tr>
</tbody>
</table>

Examples

1. Set register Rx to Ry if the LT bit is set in CR0, and to Rz otherwise.
   \[\text{isellt } Rx,Ry,Rz \quad \text{(equivalent to: } isel \quad Rx,Ry,Rz,0)\]

2. Set register Rx to Ry if the GT bit is set in CR0, and to Rz otherwise.
   \[\text{iselgt } Rx,Ry,Rz \quad \text{(equivalent to: } isel \quad Rx,Ry,Rz,1)\]

3. Set register Rx to Ry if the EQ bit is set in CR0, and to Rz otherwise.
   \[\text{iseleq } Rx,Ry,Rz \quad \text{(equivalent to: } isel \quad Rx,Ry,Rz,2)\]
E.8 Rotate and Shift Mnemonics

The Rotate and Shift instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation.

**Extract** Select a field of n bits starting at bit position b in the source register; left or right justify this field in the target register; clear all other bits of the target register to 0.

**Insert** Select a left-justified or right-justified field of n bits in the source register; insert this field starting at bit position b of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

**Rotate** Rotate the contents of a register right or left n bits without masking.

**Shift** Shift the contents of a register right or left n bits, clearing vacated bits to 0 (logical shift).

**Clear** Clear the leftmost or rightmost n bits of a register to 0.

**Clear left and shift left**

Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

E.8.1 Operations on Doublewords

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extended Mnemonic</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract and left justify immediate</td>
<td>extldi ra,rs,n,b  (n &gt; 0)</td>
<td>rldicr ra,rs,rb,n-1</td>
</tr>
<tr>
<td>Extract and right justify immediate</td>
<td>extrdi ra,rs,n,b  (n &gt; 0)</td>
<td>rldicl ra,rs,rb,n-64</td>
</tr>
<tr>
<td>Insert from right immediate</td>
<td>insrdi ra,rs,n,b  (n &gt; 0)</td>
<td>rldimi ra,rs,n,64-n,b</td>
</tr>
<tr>
<td>Rotate left immediate</td>
<td>rotldi ra,rs,n</td>
<td>rldicl ra,rs,64-n</td>
</tr>
<tr>
<td>Rotate right immediate</td>
<td>rotrdi ra,rs,n</td>
<td>rldicl ra,rs,n,64-n</td>
</tr>
<tr>
<td>Rotate left</td>
<td>rotld ra,rs,rb</td>
<td>rldcl ra,rs,rb,0</td>
</tr>
<tr>
<td>Shift left immediate</td>
<td>sldi ra,rs,n  (n &lt; 64)</td>
<td>rldicr ra,rs,n,63-n</td>
</tr>
<tr>
<td>Shift right immediate</td>
<td>srdi ra,rs,n  (n &lt; 64)</td>
<td>rldicl ra,rs,n,64-n</td>
</tr>
<tr>
<td>Clear left immediate</td>
<td>clrldi ra,rs,n  (n &lt; 64)</td>
<td>rldicl ra,rs,0-n</td>
</tr>
<tr>
<td>Clear right immediate</td>
<td>clrldi ra,rs,n  (n &lt; 64)</td>
<td>rldicl ra,rs,0,63-n</td>
</tr>
<tr>
<td>Clear left and shift left immediate</td>
<td>clrlsldi ra,rs,b,n  (n &lt;= b &lt; 64)</td>
<td>rldic ra,rs,n,b</td>
</tr>
</tbody>
</table>

Examples

1. Extract the sign bit (bit 0) of register Ry and place the result right-justified into register Rx.
   
   `extrdi Rx,Ry,1,0`  (equivalent to:  `rldicl Rx,Ry,1,63`)

2. Insert the bit extracted in (1) into the sign bit (bit 0) of register Rz.
   
   `insrdi Rz,Rx,1,0`  (equivalent to:  `rldimi Rz,Rx,63,0`)

3. Shift the contents of register Rx left 8 bits.
   
   `sldi Rx,Rx,8`  (equivalent to:  `rldicr Rx,Rx,8,55`)

4. Clear the high-order 32 bits of register Ry and place the result into register Rx.
   
   `clrldi Rx,Ry,32`  (equivalent to:  `rldicl Rx,Ry,0,32`)

717  Power ISA™ - Book I
E.8.2 Operations on Words

All these mnemonics can be coded with a final “.” to cause the Rc bit to be set in the underlying instruction. The operations as described above apply to the low-order 32 bits of the registers, as if the registers were 32-bit registers. The Insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.
   extrwi Rx,Ry,1,0 (equivalent to: rlwinm Rx,Ry,1,31,31)
2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.
   insrwi Rz,Rx,1,0 (equivalent to: rlwimi Rz,Rx,31,0,0)
3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.
   slwi Rx,Rx,8 (equivalent to: rlwinm Rx,Rx,8,0,23)
4. Clear the high-order 16 bits of the low-order 32 bits of register Ry and place the result into register Rx, clearing the high-order 32 bits of register Rx.
   clrlwi Rx,Ry,16 (equivalent to: rlwinm Rx,Ry,0,16,31)
E.9 Move To/From Special Purpose Register Mnemonics

The `mtspr` and `mfspr` instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand.

### Examples

1. Copy the contents of register Rx to the XER.
   ```assembly
   mtxer Rx  \(\text{equivalent to: } \text{mtspr } 1, Rx\)
   ```

2. Copy the contents of the LR to register Rx.
   ```assembly
   mflr Rx  \(\text{equivalent to: } \text{mfspr } Rx, 8\)
   ```

3. Copy the contents of register Rx to the CTR.
   ```assembly
   mtctr Rx  \(\text{equivalent to: } \text{mtspr } 9, Rx\)
   ```

### E.10 Miscellaneous Mnemonics

#### No-op

Many Power ISA instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that will trigger this.

```assembly
nop  \(\text{equivalent to: } \text{ori } 0, 0, 0\)
```

For some uses of a no-op instruction, optimizations related to no-ops, such as removal from the execution stream, are not desirable. An extended mnemonic is provided for the executed form of no-op. This form of no-op will still consume execution resources.

```assembly
xnop  \(\text{equivalent to: } \text{xori } 0, 0, 0\)
```

#### Load Immediate

The `addi` and `addis` instructions can be used to load an immediate value into a register. Extended mnemonics are provided to convey the idea that no addition is being performed but merely data movement (from the immediate field of the instruction to a register).

Load a 16-bit signed immediate value into register Rx.

```assembly
li Rx, value  \(\text{equivalent to: } \text{addi } Rx, 0, \text{value}\)
```

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.

```assembly
lis Rx, value  \(\text{equivalent to: } \text{addis } Rx, 0, \text{value}\)
```
**Load Address**

This mnemonic permits computing the value of a base-displacement operand, using the `addi` instruction which normally requires separate register and immediate operands.

\[
\text{la} \quad Rx,D(Ry) \quad \text{(equivalent to:} \quad \text{addi} \quad Rx,Ry,D) \\
\]

The `la` mnemonic is useful for obtaining the address of a variable specified by name, allowing the Assembler to supply the base register number and compute the displacement. If the variable v is located at offset Dv bytes from the address in register Rv, and the Assembler has been told to use register Rv as a base for references to the data structure containing v, then the following line causes the address of v to be loaded into register Rx.

\[
\text{la} \quad Rx,v \quad \text{(equivalent to:} \quad \text{addi} \quad Rx,Rv,Dv) \\
\]

**Move Register**

Several Power ISA instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Ry to register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

\[
\text{mr} \quad Rx,Ry \quad \text{(equivalent to:} \quad \text{or} \quad Rx,Ry,Ry) \\
\]

**Complement Register**

Several Power ISA instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Ry and places the result into register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

\[
\text{not} \quad Rx,Ry \quad \text{(equivalent to:} \quad \text{nor} \quad Rx,Ry,Ry) \\
\]

**Move To/From Condition Register**

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the Condition Register, using the same style as the `mfcr` instruction.

\[
\text{mtcr} \quad Rx \quad \text{(equivalent to:} \quad \text{mtcrf} \quad 0xFF,Rx) \\
\]

The following instructions may generate either the (old) `mtcrf` or `mfcr` instructions or the (new) `mtocrf` or `mfocrf` instruction, respectively, depending on the target machine type assembler parameter.

\[
\text{mtcrf} \quad FXM,Rx \\
\text{mfcr} \quad Rx \\
\]

All three extended mnemonics in this subsection are being phased out. In future assemblers the form "mtcr Rx" may not exist, and the `mtcrf` and `mfcr` mnemonics may generate the old form instructions (with bit 11 = 0) regardless of the target machine type assembler parameter, or may cease to exist.
Appendix F. Programming Examples

F.1 Multiple-Precision Shifts

This section gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is defined to be a shift of an N-doubleword quantity (64-bit mode) or an N-word quantity (32-bit mode), where N>1. The quantity to be shifted is contained in N registers. The shift amount is specified either by an immediate value in the instruction, or by a value in a register.

The examples shown below distinguish between the cases N=2 and N>2. If N=2, the shift amount may be in the range 0 through 127 (64-bit mode) or 0 through 63 (32-bit mode), which are the maximum ranges supported by the Shift instructions used. However if N>2, the shift amount must be in the range 0 through 63 (64-bit mode) or 0 through 31 (32-bit mode), in order for the examples to yield the desired result. The specific instance shown for N>2 is N=3; extending those code sequences to larger N is straightforward, as is reducing them to the case N=2 when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case N=3 is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit mode for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR 6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 30 and 31 are used as scratch registers.

For N>2, the number of instructions required is 2N–1 (immediate shifts) or 3N–1 (non-immediate shifts).
Multiple-precision shifts in 64-bit mode [Category: 64-Bit]

**Shift Left Immediate, N = 3 (shift amnt < 64)**
- `rdicr r5,r4,sh,63-sh`
- `rdimi r4,r3,0,sh`
- `rdicl r4,r4,sh,0`
- `rdimi r3,r2,0,sh`
- `rdicl r3,r3,sh,0`

**Shift Left, N = 2 (shift amnt < 128)**
- `subfic r31,r6,64`
- `sl d r2,r2,r6`
- `sr d r30,r3,r31`
- `or r2,r2,r30`
- `addi r31,r6,−64`
- `sl d r3,r3,r6`
- `sr d r4,r4,r6`

**Shift Left, N = 3 (shift amnt < 64)**
- `subfic r31,r6,64`
- `sl d r2,r2,r6`
- `sr d r30,r3,r31`
- `or r2,r2,r30`
- `sl d r3,r3,r6`
- `sr d r30,r4,r31`
- `or r3,r3,r30`
- `sl d r4,r4,r6`

**Shift Right Immediate, N = 3 (shift amnt < 64)**
- `rdimi r4,r3,0,64-sh`
- `rdicl r4,r4,64-sh,0`
- `rdimi r3,r2,0,64-sh`
- `rdicl r3,r3,64-sh,0`
- `rdicl r2,r2,64-sh,sh`

**Shift Right, N = 2 (shift amnt < 128)**
- `subfic r31,r6,64`
- `sr d r3,r3,r6`
- `sl d r30,r2,r31`
- `or r3,r3,r30`
- `addi r31,r6,−64`
- `sr d r30,r2,r31`
- `or r3,r3,r30`
- `sr d r2,r2,r6`

**Shift Right, N = 3 (shift amnt < 64)**
- `subfic r31,r6,64`
- `sr d r4,r4,r6`
- `sl d r30,r3,r31`
- `or r4,r4,r30`
- `sr d r3,r3,r6`
- `sl d r30,r2,r31`
- `or r3,r3,r30`
- `sr d r2,r2,r6`

Multiple-precision shifts in 32-bit mode

**Shift Left Immediate, N = 3 (shift amnt < 32)**
- `rlwinm r2,r2,sh,0,31-sh`
- `rlwimi r2,r3,sh,32-sh,31`
- `rlwinm r3,r3,sh,0,31-sh`
- `rlwimi r3,r4,sh,32-sh,31`
- `rlwinm r4,r4,sh,0,31-sh`

**Shift Left, N = 2 (shift amnt < 128)**
- `subfic r31,r6,32`
- `sl w r2,r2,r6`
- `sn w r30,r3,r31`
- `or r2,r2,r30`
- `addi r31,r6,−32`
- `sl w r30,r3,r31`
- `or r2,r2,r30`
- `sl w r3,r3,r6`

**Shift Left, N = 3 (shift amnt < 32)**
- `subfic r31,r6,32`
- `sl w r2,r2,r6`
- `sn w r30,r3,r31`
- `or r2,r2,r30`
- `sl w r3,r3,r6`
- `sn w r30,r4,r31`
- `or r3,r3,r30`
- `sl w r4,r4,r6`

**Shift Right Immediate, N = 3 (shift amnt < 32)**
- `rlwinm r4,r4,32-sh,sh,31`
- `rlwimi r4,r3,32-sh,0,sh−1`
- `rlwinm r3,r3,32-sh,sh,31`
- `rlwimi r3,r2,32-sh,0,sh−1`
- `rlwinm r2,r2,32-sh,sh,31`

**Shift Right, N = 2 (shift amnt < 128)**
- `subfic r31,r6,32`
- `sn w r3,r3,r6`
- `sl w r30,r2,r31`
- `or r3,r3,r30`
- `addi r31,r6,−32`
- `sn w r30,r2,r31`
- `or r3,r3,r30`
- `sn w r2,r2,r6`

**Shift Right, N = 3 (shift amnt < 32)**
- `subfic r31,r6,32`
- `sn w r4,r4,r6`
- `sl w r30,r3,r31`
- `or r4,r4,r30`
- `sn w r3,r3,r6`
- `sl w r30,r2,r31`
- `or r3,r3,r30`
- `sn w r2,r2,r6`
Multiple-precision shifts in 64-bit mode, continued [Category: 64-Bit]

Shift Right Algebraic Immediate, N = 3 (shift amnt < 64)

| rldimi r4,r3,0,64-sh |
| rdicl r4,r4,64-sh,0 |
| rdicl r3,r2,0,64-sh |
| rdicl r3,r3,64-sh,0 |
| sradi r2,r2,sh |

Shift Right Algebraic, N = 2 (shift amnt < 128)

| subfic r31,r6,64 |
| srd r3,r3,r6 |
| sld r30,r2,r31 |
| or r3,r3,r30 |
| addic. r31,r6,-64 |
| srad r30,r2,r31 |
| isel r3,r30,r3,gt |
| srad r2,r2,r6 |

Shift Right Algebraic, N = 3 (shift amnt < 64)

| subfic r31,r6,64 |
| srd r4,r4,r6 |
| sld r30,r3,r31 |
| or r4,r4,r30 |
| srd r3,r3,r6 |
| sld r30,r2,r31 |
| or r3,r3,r30 |
| srad r2,r2,r6 |

Multiple-precision shifts in 32-bit mode, continued

Shift Right Algebraic Immediate, N = 3 (shift amnt < 32)

| rlwinm r4,r4,32-sh,sh,31 |
| rlwimi r4,r3,32-sh,0,sh-1 |
| rlwinm r3,r3,32-sh,sh,31 |
| rlwimi r3,r2,32-sh,0,sh-1 |
| srawi r2,r2,sh |

Shift Right Algebraic, N = 2 (shift amnt < 64)

| subfic r31,r6,32 |
| srd r3,r3,r6 |
| slw r30,r2,r31 |
| or r3,r3,r30 |
| addic. r31,r6,-32 |
| sraw r30,r2,r31 |
| isel r3,r30,r3,gt |
| sraw r2,r2,r6 |

Shift Right Algebraic, N = 3 (shift amnt < 32)

| subfic r31,r6,32 |
| srd r4,r4,r6 |
| slw r30,r3,r31 |
| or r4,r4,r30 |
| srd r3,r3,r6 |
| slw r30,r2,r31 |
| or r3,r3,r30 |
| sraw r2,r2,r6 |
F.2 Floating-Point Conversions [Category: Floating-Point]

This section gives examples of how the Floating-Point Conversion instructions can be used to perform various conversions.

Warning: Some of the examples use the fsel instruction. Care must be taken in using fsel if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section F.3.4, “Notes” on page 730.

F.2.1 Conversion from Floating-Point Number to Floating-Point Integer

The full convert to floating-point integer function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1 and the result is returned in FPR 3.

\[
\begin{align*}
\text{mtfsb0} & \ 23 \quad \text{#clear VXCVI} \\
fctid[z] & \ f3,f1 \quad \text{#convert to fx int} \\
fctid & \ f3,f3 \quad \text{#convert back again} \\
mcrfs & \ 7,5 \quad \text{#VXCVI to CR} \\
bf & \ 31,8 \quad \text{#skip if VXCVI was 0} \\
fmr & \ f3,f1 \quad \text{#input was fp int}
\end{align*}
\]

F.2.2 Conversion from Floating-Point Number to Signed Fixed-Point Integer Doubleword

The full convert to signed fixed-point integer doubleword function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value \(2^{64} - 2^{2048}\) is in FPR 3, the value \(2^{63}\) is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

\[
\begin{align*}
\text{fsel} & \ f2,f1,f1,f0 \quad \text{#use 0 if < 0} \\
\text{fsub} & \ f5,f3,f1 \quad \text{#use max if > max} \\
\text{fsel} & \ f2,f5,f2,f3 \\
\text{fcmpu} & \ cr2,f2,f4 \quad \text{#subtract } 2^{63} \\
\text{fctid[z]} & \ f2,f2 \quad \text{#use diff if }\geq 2^{63} \\
\text{stfd} & \ f2,\text{disp}(\text{r1}) \quad \text{#store float} \\
\text{ld} & \ r3,\text{disp}(\text{r1}) \quad \text{#load dword} \\
\text{blt} & \ cr2,8 \quad \text{#add } 2^{63} \text{ if input} \\
\text{add} & \ r3,r3,r4 \quad \text{# was }\geq 2^{63}
\end{align*}
\]

F.2.3 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Doubleword

The full convert to unsigned fixed-point integer doubleword function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value \(2^{64} - 2^{2048}\) is in FPR 3, the value \(2^{63}\) is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

\[
\begin{align*}
\text{fsel} & \ f2,f1,f1,f0 \quad \text{#use 0 if < 0} \\
\text{fsel} & \ f5,f3,f1 \quad \text{#use max if > max} \\
\text{fsel} & \ f2,f5,f2,f3 \\
\text{fsel} & \ f5,f2,f4 \quad \text{#subtract } 2^{63} \\
\text{fctid[z]} & \ f2,f2 \quad \text{#convert to fx int} \\
\text{stfd} & \ f2,\text{disp}(\text{r1}) \quad \text{#store float} \\
\text{ld} & \ r3,\text{disp}(\text{r1}) \quad \text{#load dword} \\
\text{blt} & \ cr2,8 \quad \text{#add } 2^{63} \text{ if input} \\
\text{add} & \ r3,r3,r4 \quad \text{# was }\geq 2^{63}
\end{align*}
\]

F.2.4 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full convert to signed fixed-point integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

\[
\begin{align*}
\text{fctiw[z]} & \ f2,f1 \quad \text{#convert to fx int} \\
\text{stfd} & \ f2,\text{disp}(\text{r1}) \quad \text{#store float} \\
\text{lwa} & \ r3,\text{disp}+4(\text{r1}) \quad \text{#load word algebraic}
\end{align*}
\]
F.2.5 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

The full convert to unsigned fixed-point integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value $2^{32} - 1$ is in FPR 3, and the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```assembly
fsel f2,f1,f0,f0 # use 0 if < 0
fsub f4,f3,f1 # use max if > max
fsel f2,f4,f2,f3
fctid[f] f2,f2 # convert to fx int
stfd f2,disp(r1) # store float
```

An alternative, shorter, sequence can be used if rounding according to FSCPR is desired and FPSCR specifies Round toward +Infinity or Round toward -Infinity, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given fixed-point integer. In this case the full convert from unsigned fixed-point integer doubleword function can be implemented with the sequence shown below, assuming the value $2^{64}$ is in FPR 2.

```assembly
std r3,disp(r1) # store dword
lfd f1,disp(r1) # load float
fcfid f1,f1 # convert to fp int
```

F.2.6 Conversion from Signed Fixed-Point Integer Doubleword to Floating-Point Number

The full convert from signed fixed-point integer doubleword function, using the rounding mode specified by FPSCR can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```assembly
std r3,disp(r1) # store dword
lfd f1,disp(r1) # load float
fcfid f1,f1 # convert to fp int
```

F.2.7 Conversion from Unsigned Fixed-Point Integer Doubleword to Floating-Point Number

The full convert from unsigned fixed-point integer doubleword function, using the rounding mode specified by FPSCR, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the value $2^{32}$ is in FPR 4, and the result is returned in FPR 1, and two doublewords at displacement “disp” from the address in GPR 1 can be used as scratch space.

```assembly
rldicl r2,r3,32,32 # isolate high half
rldicl r0,r3,0,32 # isolate low half
std r2,disp(r1) # store dword both
std r0,disp+8(r1)
lfd f2,disp(r1) # load float both
lfd f1,disp+8(r1)
fcfid f2,f2 # convert each half to
fcfid f1,f1 # fp int (exact result)
fmadd f1,f4,f2,f1 # \((2^{32})\times high + low\)
```

F.2.8 Conversion from Signed Fixed-Point Integer Word to Floating-Point Number

The full convert from signed fixed-point integer word function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space. (The result is exact.)

```assembly
extsw r3,r3 # extend sign
std r3,disp(r1) # store dword
lfd f1,disp(r1) # load float
fcfid f1,f1 # convert to fp int
```

The following sequence can be used, assuming a word at the address in GPR 1 + GPR 2 can be used as scratch space.

```assembly
stwx r3,r1,r2 # store word
lfiwax f1,r1,r2 # load float
fcfid f1,f1 # convert to fp int
```

F.2.9 Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number

The full convert from unsigned fixed-point integer word function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space. (The result is exact.)

```assembly
rldicl r0,r3,0,32 # zero-extend
std r0,disp(r1) # store dword
lfd f1,disp(r1) # load float
fcfid f1,f1 # convert to fp int
```
### F.2.10 Unsigned Single-Precision BCD Arithmetic

The `addg6s` instruction can be used to add or subtract two BCD operands. In these examples it is assumed that `r0` contains `0x666...666`. (BCD data formats are described in Section 6.3 of Book I.)

Addition of the unsigned BCD operand in register `RA` to the unsigned BCD operand in register `RB` can be accomplished as follows.

```
add r1,RA,r0
add r2,r1,RB
addg6s RT,r1,RB # RT = RA + BCD RB
```

Subtraction of the unsigned BCD operand in register `RA` from the unsigned BCD operand in register `RB` can be accomplished as follows. (In this example it is assumed that `RB` is not register 0.)

```
addi r1,RB,1
nor r2,RA,RA  # one's complement of RA
add r3,r1,r2
addg6s RT,r1,r2 # RT = RA - BCD RB
```

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).

### F.2.11 Signed Single-Precision BCD Arithmetic

Addition of the signed 15-digit BCD operand in register `RA` to the signed BCD operand in register `RB` can be accomplished as follows. If the signs of operands are different, then the operand of smaller magnitude is subtracted from the operand of larger magnitude and the sign of the larger operand is preserved; otherwise the operands are added and the sign is preserved.

The sign code is in the low order 4 bits of the operands and uses one of the standard encodings. (See Section 6.3 of Book I for a description of BCD and sign encodings.) This example assumes preferred sign option 1 (0b1100 is plus and 0b1101 is minus). For preferred sign option 2 (0b1111 is plus and 0b1101 is minus), replace the `xori` after the "SignedSub" label with "`xori RA,RA,1".

```
xori RA,RA,1
```

SignedAdd:
```
xor r5,RA,RB
andi. r5,r5,15  # compare sign codes
cmpld cr1,RA,RB  # compare magnitudes
beq cr0,samesign
bl cr1,EminusA
```

# set up for RT = RA - BCD RB
```
nor r9,RB,RB    # one's complement of RB
addi r10,RA,16 # generate the carry in
b submag
```

EminusA:
```
# set up for RT = RB - BCD RA
nor r9,RA,RA    # one's complement of RA
addi r10,RB,16 # generate the carry in
```

submag:
```
    rldicr r9,RA,0,59  # remove the sign code
add     r8,RA,0
addg6s  RT,RA,0     # sign position
subf    RT,RT,r8    # RT = RB - BCD RA
b       done
```

samesign:
```
    rldicr r8,RB,0,59  # remove the sign code
add     r10,RA,11
add     r9,RB,8
addg6s  RT,RB       # add 6's
subf    RT,RT,r9    # RT = RA + BCD RB
```

done:

### F.2.12 Unsigned Extended-Precision BCD Arithmetic

Multiple precision BCD arithmetic requires additional code to add/subtract higher order digits and handle the carry between 16 digit groups. For example, the following sequence implements a 32-digit BCD add. In this example the contents of register `R3` concatenated with the contents of `R4` represent the first 32-digit operand and the contents of register `R5` concatenated with the contents of `R6` represents the second operand. The contents of register `R3` concatenated with the contents of register `R4` represents the result.

(In this example `r0` contains `0x6666 6666 6666 6666`.)

```
    add r10,R4,r0
    addc r9,R10,R6  # generate the carry
    addg6s R4,R10,R6 # RT1 = RA1 + BCD RB1
```

```
    rldicr r8,RB,0,59  # remove the sign code
add     r10,RA,11
add     r9,RB,8
addg6s  RT,RB       # add 6's
subf    RT,RT,r9    # RT = RA + BCD RB0
```

Preserving the appropriate sign code is accomplished by zeroing the sign code of the other operand before performing a 16 digit BCD addition/subtraction. Other addends (ones complement or 6's) must leave the sign code position as zero.

(In this example `r11` contains `0x6666 6666 6666 6666`.)

SignedSub:
```
xori RA,RA,1
```

```
add r10,R4,r0
addc r9,R10,R6  # generate the carry
addg6s R4,R10,R6 # RT1 = RA1 + BCD RB1
```

```
addze R5,R5      # propagate the carry
add r10,R3,r0
add r9,R10,R5
addg6s R3,R10,R5 # RT0 = RA0 + BCD RB0
```

```
```
Note that an extra instruction (\textit{addze}) is required to propagate the carry so that the same value is used in the subsequent \textit{add} and \textit{addg6s}.

The following sequence implements a 32-digit BCD subtraction. In this example the first operand in R3 and R4 is subtracted from the 2nd operand in R5 and R6. The result is in R3 and R4.

\begin{verbatim}
addi r10,R6,1
nor r9,R4,R4  # one's complement of RAO
addc r8,r10,r9 # Generate the carry
addg6s R4,r10,r9
subf R4,R4,r8 # RT1 = RB1 - BCD RA1
addze r10,R5 # propagate the carry
nor r9,R3,R3  # one's complement of RAO
add r8,r10,r2
addg6s R3,r10,r9
subf R3,R3,r8 # RT0 = RB0 - BCD RA0
\end{verbatim}
F.3 Floating-Point Selection [Category: Floating-Point]

This section gives examples of how the `Floating Select` instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using `fsel` and other Power ISA instructions. In the examples, a, b, x, y, and z are floating-point variables, which are assumed to be in FPRs fa, fb, fx, fy, and fz. FPR fs is assumed to be available for scratch space.

Additional examples can be found in Section F.2, “Floating-Point Conversions [Category: Floating-Point]” on page 726.

Warning: Care must be taken in using `fsel` if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section F.3.4.

F.3.1 Comparison to Zero

<table>
<thead>
<tr>
<th>High-level language:</th>
<th>Power ISA:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a ≥ 0.0 then x ← y else x ← z</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(1)</td>
</tr>
<tr>
<td>if a &gt; 0.0 then x ← y else x ← z</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(1,2)</td>
</tr>
<tr>
<td>if a = 0.0 then x ← y else x ← z</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(1)</td>
</tr>
</tbody>
</table>

F.3.2 Minimum and Maximum

<table>
<thead>
<tr>
<th>High-level language:</th>
<th>Power ISA:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← min(a,b)</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(3,4,5)</td>
</tr>
<tr>
<td>x ← max(a,b)</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(3,4,5)</td>
</tr>
</tbody>
</table>

F.3.3 Simple if-then-else Constructions

<table>
<thead>
<tr>
<th>High-level language:</th>
<th>Power ISA:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a ≥ b then x ← y else x ← z</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(4,5)</td>
</tr>
<tr>
<td>if a &gt; b then x ← y else x ← z</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(3,4,5)</td>
</tr>
<tr>
<td>if a = b then x ← y else x ← z</td>
<td>fsub fs,fa,fb,fa fsub fx,fx,fa,fy fsub fx,fx,fa,fy</td>
<td>(4,5)</td>
</tr>
</tbody>
</table>

F.3.4 Notes

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (,<, ≤, and ≠). They should also be considered when any other use of `fsel` is contemplated.

In these Notes, the “optimized program” is the Power ISA program shown, and the “unoptimized program” (not shown) is the corresponding Power ISA program that uses `fcmpu` and `Branch Conditional` instructions instead of `fsel`.

1. The unoptimized program affects the VXSNAN bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.

2. The optimized program gives the incorrect result if a is a NaN.

3. The optimized program gives the incorrect result if a and/or b is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).

4. The optimized program gives the incorrect result if a and b are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)

5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.
F.4 Vector Unaligned Storage Operations [Category: Vector]

F.4.1 Loading a Unaligned Quadword Using Permute from Big-Endian Storage

The following sequence of instructions copies the unaligned quadword storage operand into VRT.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb  # load MSQ
lvsl   Vp,0,Rb   # set permute control vector
addi   Rb,Rb,16 # address of LSQ
lvx    Vlo,0,Rb  # load LSQ
perm   Vt,Vhi,Vlo,Vp # align the data
```
Book II:

Power ISA Virtual Environment Architecture
Chapter 1. Storage Model

1.1 Definitions

The following definitions, in addition to those specified in Book I, are used in this Book. In these definitions, “Load instruction” includes the Cache Management and other instructions that are stated in the instruction descriptions to be “treated as a Load”, and similarly for “Store instruction”.

- system
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the privileged software.

- main storage
  The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.

- primary cache
  The level of cache closest to the processor.

- secondary cache
  After the primary cache, the next closest level of cache to the processor.

- instruction storage
  The view of storage as seen by the mechanism that fetches instructions.

- data storage
  The view of storage as seen by a Load or Store instruction.

- program order
  The execution of instructions in the order required by the sequential execution model. (See Section 2.2 of Book I.) A dcbz instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a Store instruction as described there. An additional exception to the sequential execution model beyond those described in Book I is caused by transaction failure (see Section 5.3.3).

- storage location
  A contiguous sequence of one or more bytes in storage. When used in association with a specific instruction or the instruction fetching mechanism, the length of the sequence of one or more bytes is typically implied by the operation. In other uses, it may refer more abstractly to a group of bytes which share common storage attributes.

- storage access
  An access to a storage location. There are three (mutually exclusive) kinds of storage access.

  - data access
    An access to the storage location specified by a Load or Store instruction, or, if the access is performed “out-of-order” (see Section 5.5 of Book III-S and Section 6.5 of Book III-E), an access to a storage location as if it were the storage location specified by a Load or Store instruction.

  - instruction fetch
    An access for the purpose of fetching an instruction.

  - implicit access
    An access by the processor for the purpose of address translation or reference and change recording (see Book III-S).

- caused by, associated with

  - caused by
    A storage access is said to be caused by an instruction if the instruction is a Load or Store and the access (data access) is to the storage location specified by the instruction.

  - associated with
    A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.
prefetched instructions
Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.

uniprocessor
A system that contains one processor.

multiprocessor
A system that contains two or more processors.

shared storage multiprocessor
A multiprocessor that contains some common storage, which all the processors in the system can access.

performed
A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when an instruction fetch by P2 will not be satisfied from the copy of the block that existed in its instruction cache when the instruction causing the invalidation was executed, and similarly for a data cache block invalidation.

The preceding definitions apply regardless of whether P1 and P2 are the same entity.

page (virtual page)
$2^n$ contiguous bytes of storage aligned such that the effective address of the first byte in the page is an integral multiple of the page size for which protection and control attributes are independently specifiable and for which reference and change status <$S$> are independently recorded.

block
The aligned unit of storage operated on by the Cache Management instructions. The size of an instruction cache block may differ from the size of a data cache block, and both sizes may vary between implementations. The maximum block size is equal to the minimum page size.

aligned storage access
A load or store is aligned if the address of the target storage location is a multiple of the size of the transfer effected by the instruction.

aggregate store
The set of stores caused by a successful transaction, which are performed as an atomic unit.

1.2 Introduction

The Power ISA User Instruction Set Architecture, discussed in Book I, defines storage as a linear array of bytes indexed from 0 to a maximum of $2^{64}$-1. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The Power ISA Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The Power ISA Virtual Environment Architecture, in conjunction with services based on the Power ISA Operating Environment Architecture (see Book III) and provided by the operating system, permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the Power ISA specifies a relaxed model of storage consistency. In a multiprocessor system that allows multiple copies of a storage location, aggressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the Power ISA that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The Power ISA system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a “virtual” address space larger than either the effective address space or the real address space.

Each program can access $2^{64}$ bytes of “effective address” (EA) space, subject to limitations imposed by the operating system. In a typical Power ISA system, each program's EA space is a subset of a larger “virtual address” (VA) space managed by the operating system. Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the
operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, "paging" activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access; see Book III), based on system standards (e.g., virtual pages (including read/write, read only, and no access); see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-Copy Atomicity

An access is single-copy atomic, or simply atomic, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

Vector storage accesses are not guaranteed to be atomic. The following other types of single-register accesses are always atomic:
- byte accesses (all bytes are aligned on byte boundaries)
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries
- doubleword accesses aligned on doubleword boundaries (64-bit implementations only; see Section 1.2 of Book III-E)<E>

No other accesses are guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic.
- any Load or Store instruction for which the operand is unaligned
- imw, stmw, lwisi, lswi, stswi, stswx
- lfdp, lfdpx, stfdp, stfdpx
- any Cache Management instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. In general, the number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed. The only exception to the preceding rule is that, for lfdp, lfdpx, stfdp, and stfdpx, if the access is aligned on a doubleword boundary, it is performed as a pair of disjoint atomic doubleword accesses.

The results for several combinations of loads and stores to the same or overlapping locations are described below.
1. When two processors execute atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.
2. When two processors execute atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
3. When two processors execute stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors execute stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
5. When a processor executes an atomic store to a location, a second processor executes an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location before the store or the contents of the location after the store.
6. When a load and a store with the same target location can be executed simultaneously, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

1.5 Cache Model

A cache model in which there is one cache for instructions and another cache for data is called a "Harvard-style" cache. This is the model assumed by the Power ISA, e.g., in the descriptions of the Cache Management instructions in Section 4.3. Alternative cache models may be implemented (e.g., a "combined cache" model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (e.g., modifications caused by Store instructions).

A location in the data cache is considered to be modified in that cache if the location has been modified (e.g., by a Store instruction) and the modified data have not been written to main storage.

Cache Management instructions are provided so that programs can manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that will be executed (i.e., when the program modifies...
data in storage and then attempts to execute the modified data as instructions. The Cache Management instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage control attributes associated with the specified storage location (see Section 1.6, “Storage Control Attributes”).

The Cache Management instructions allow the program to do the following.

- invalidate the copy of storage in an instruction cache block (icbi)
- provide a hint that an instruction will probably soon be accessed from a specified instruction cache block (icbi)
- provide a hint that the program will probably soon access a specified data cache block (dcbt, dcbtst)
- <E> allocate a data cache block and set the contents of that block to zeros, but perform no operation if no write access is allowed to the data cache block (dcbf)
- set the contents of a data cache block to zeros (dcbz)
- copy the contents of a modified data cache block to main storage (dcbs)
- copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (dcbf or dcbt)

1.6 Storage Control Attributes

Some operating systems may provide a means to allow programs to specify the storage control attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write Through Required
- Caching Inhibited
- Memory Coherence Required
- Guarded
- Endianness <E>
- Strong Access Order [Category: SAO]

These attributes have meaning only when an effective address is translated by the processor performing the storage access.

<E> Additional storage control attributes may be defined for some implementations. See Section 6.8 of Book III-E for additional information.

---

Programming Note

The Write Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the lbarx, lbarx, lbarx, ldarx, lqarx, stbcx, sthcx, stwcx, stdcx, and stqcx instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes.

In the remainder of this section, “Load” instruction includes the Cache Management and other instructions that are stated in the instruction descriptions to be “treated as a Load” unless they are explicitly excluded, and similarly for “Store” instruction.

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A Store instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate Store instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the Store instructions are separated by a sync, eieio <S>, or mbar <E> instruction.

1.6.2 Caching Inhibited

An access to a Caching Inhibited storage location is performed in main storage. A Load instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate Load instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate Store instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the Load or Store instructions are separated by a sync or mbar <E> instruction. Combining may also occur.
among such accesses from multiple processors that share a common memory interface. No combining occurs if the storage is also Guarded.

Programming Note

None of the memory barrier instructions prevent the combining of accesses from different processors. The Guarded storage attribute must be used in combination with Caching Inhibited to prevent such combining.

1.6.3 Memory Coherence Required [Category: Memory Coherence]

An access to a Memory Coherence Required storage location is performed coherently, as follows.

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are coherent if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a “newer” value first and then, later, load an “older” value.

Memory coherence is managed in blocks called coherence blocks. Their size is implementation-dependent, but is larger than a word and is usually the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in this document it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

Programming Note

Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the sync instruction, the Cache Management instructions, and services provided by the operating system.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

For the Server environment, instructions are not fetched from virtual storage that is Guarded. If the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler may be invoked (see Section 6.5.5 of Book III-S).

Programming Note

In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not “well-behaved” (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).
1.6.5 Endianness [Category: Embedded.Little-Endian]

The Endianness storage control attribute specifies the byte ordering (Big-Endian or Little-Endian) that is used when the storage location is accessed; see Section 1.10 of Book I.

1.6.6 Variable Length Encoded (VLE) Instructions

VLE storage is used to store VLE instructions. Instructions fetched from VLE storage are processed as VLE instructions. VLE storage must also be Big-Endian. Instructions fetched from VLE storage that is Little-Endian cause a Byte-ordering exception, and the system instruction storage error handler will be invoked.

The VLE attribute has no effect on data accesses. See Chapter 1 of Book VLE.
1.6.7 Strong Access Order [Category: SAO]

All accesses to storage with the Strong Access Order (SAO) attribute (referred to as SAO storage) will be performed using a set of ordering rules different from that of the weakly consistent model that is described in Section 1.7.1, “Storage Access Ordering”. These rules apply only to accesses that are caused by a Load or a Store, and not to accesses associated with those instructions. Furthermore, these rules do not apply to accesses that are caused by or associated with instructions that are stated in their descriptions to be “treated as a Load” or “treated as a Store.” The details are described below, from the programmer’s point of view. (The processor may deviate from these rules if the programmer cannot detect the deviation.) The SAO attribute is not intended to be used for general purpose programming. It is provided in a manner that is not fully independent of the other storage attributes. Specifically, it is only provided for storage that is Memory Coherence Required, but not Write Through Required, not Caching Inhibited, and not Guarded. See Section 5.8.2.1, “Storage Control Bit Restrictions”, in Book III-S for more details. Accesses to SAO storage are likely to be performed more slowly than similar accesses to non-SAO storage.

The order in which a processor performs storage accesses to SAO storage, the order in which those accesses are performed with respect to other processors and mechanisms, and the order in which those accesses are performed in main storage are the same except in the circumstances described in the following paragraph. The ordering rules for accesses performed by a single processor to SAO storage are as follows.

Stores are performed in program order. When a store accesses data adjacent to that which is accessed by the next store in program order, the two storage accesses may be combined into a single larger access.

Loads are performed in program order. When a load accesses data adjacent to that which is accessed by the next load in program order, the two storage accesses may be combined into a single larger access.

Stores may not be performed before loads which precede them in program order. Loads may be performed before stores which precede them in program order, with the provision that a load which follows a store of the same datum (to the same address) must obtain a value which is no older (in consideration of the possibility of programs on other processors sharing the same storage) than the value stored by the preceding store.

When any given processor loads the datum it just stored, as described above, the load may be performed by the processor before the preceding store has been performed with respect to other processors and mechanisms, and in main storage. This may cause the processor to see its store earlier relative to stores performed by other processors than it is observed by other processors and mechanisms, and than it is performed in memory. A direct consequence of this consideration is that although programs running on each processor will see the same sequence of accesses from any individual processor to SAO storage, each may in general see a different interleaving of the individual sequences. The memory barrier instructions may be used to establish stronger ordering, as described in Section 1.7.1, “Storage Access Ordering”, beginning with the third major bullet.
1.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

1.7.1 Storage Access Ordering

The Power ISA defines two models for the ordering of storage accesses: weakly consistent and strong access ordering. The predominant model is weakly consistent. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when storage is shared by two or more programs. Implementations which support Category SAO apply a stronger consistency model among accesses to SAO storage. The order between accesses to SAO storage and those performed using the weakly consistent model is characteristic of the weakly consistent model. The following description, through the second major bullet, applies only to the weakly consistent model. The corresponding description for SAO storage is found in Section 1.6.7, “Strong Access Order [Category: SAO]”. The rest of the description following the second bulletted item applies to both models.

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase “to the extent required by the associated Memory Coherence Required attributes” refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two Store instructions or two Load instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.

- If a Load instruction depends on the value returned by a preceding Load instruction (because the value is used to compute the effective address specified by the second Load), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first Load is ANDed with zero and then added to the effective address specified by the second Load).

- When a processor (P1) executes a Synchronize, eieio<->, or mbar<-> instruction a memory barrier is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair a_i,b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be “cumulative” if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

- A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.

- B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e., by an instruction for which the access is not atomic), even if the accesses are to SAO storage, and no means are provided for controlling that order.
Because stores cannot be performed “out-of-order” (see Book III), if a Store instruction depends on the value returned by a preceding Load instruction (because the value returned by the Load is used to compute either the effective address specified by the Store or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if whether the Store instruction is executed depends on a conditional Branch instruction that in turn depends on the value returned by a preceding Load instruction.

Because an isync instruction prevents the execution of instructions following the isync until instructions preceding the isync have completed, if an isync follows a conditional Branch instruction that depends on the value returned by a preceding Load instruction, the load on which the Branch depends is performed before any loads caused by instructions following the isync. This applies even if the effects of the “dependency” are independent of the value loaded (e.g., the value is compared to itself and the Branch tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a Load instruction specifies the same storage location as a preceding Store instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a “store queue” (a buffer into which the processor places stored values before presenting them to the storage sub-system), and not be visible to other processors and mechanisms. A consequence is that if a subsequent Store depends on the value returned by the Load, the two stores need not be performed in program order with respect to other processors and mechanisms.

- Because a Store Conditional instruction may complete before its store has been performed, a conditional Branch instruction that depends on the CR0 value set by a Store Conditional instruction does not order the Store Conditional’s store with respect to storage accesses caused by instructions that follow the Branch.

- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement nonarchitected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two Load instructions) do not order storage accesses.

Examples of correct uses of dependencies, sync, lwsync, and eieio S to order storage accesses can be found in Appendix B. “Programming Examples for Sharing Storage” on page 829.

Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before storage accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same storage location, if the location is in storage that is Memory Coherence Required the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is in storage that is Caching Inhibited.

Because accesses to storage that is Caching Inhibited are performed in main storage, memory barriers and dependencies on Load instructions order such accesses with respect to any processor or mechanism even if the storage is not Memory Coherence Required.
1.7.2 Storage Ordering of I/O Accesses

A “coherence domain” consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

1.7.3 Atomic Update

The Load And Reserve and Store Conditional instructions together permit atomic update of a shared storage location. There are byte, halfword, word, doubleword, and quadword forms of each of these instructions. Described here is the operation of the word forms lwarx and stwcx; operation of the byte, halfword, dou-
A reservation for a subsequent \texttt{stwcx}. instruction is created.

2. The memory coherence mechanism is notified that a reservation exists for the storage location specified by the \texttt{lwarx}.

The \texttt{stwcx}. instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the \texttt{lwarx} and on whether the same storage location is specified by both instructions. To emulate an atomic operation with these instructions, it is necessary that both the \texttt{lwarx} and the \texttt{stwcx}. specify the same storage location.

A \texttt{stwcx}. performs a store to the target storage location only if the storage location specified by the \texttt{lwarx} that established the reservation has not been stored into by another processor or mechanism since the reservation was created. If the storage locations specified by the two instructions differ, the store is not necessarily performed except that if the Store Conditional Page Mobility category is supported and the storage locations are in different naturally aligned blocks of real storage whose size is the smallest real page size supported by the implementation, the store is not performed.

A \texttt{stwcx}. that performs its store is said to “succeed”.

Examples of the use of \texttt{lwarx} and \texttt{stwcx}. are given in Appendix B. “Programming Examples for Sharing Storage” on page 829.

A successful \texttt{stwcx}. to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or \texttt{lwarx} from the given location by another processor may return a “stale” value. However, a subsequent \texttt{lwarx} from the given location by the other processor followed by a successful \texttt{stwcx}. by that processor is guaranteed to have returned the value stored by the first processor’s \texttt{stwcx}. (in the absence of other stores to the given location).

If a \textit{Store Conditional} instruction is used with a preceding \textit{Load And Reserve} instruction that has a different storage operand length (e.g., \texttt{stwcx}. with \texttt{idarx}), the reservation is cleared and it is undefined whether the store is performed.

\begin{center}
\textbf{Programming Note}
\end{center}

The store caused by a successful \texttt{stwcx}. is ordered, by a dependence on the reservation, with respect to the load caused by the \texttt{lwarx} that established the reservation, such that the two storage accesses are performed in program order with respect to any processor or mechanism.

Before reassigning a virtual address to a different real page, privileged software may need to clear all processors’ reservations for the original real page in order to avoid a \textit{Store Conditional} being successful only because the corresponding reservation for the original location is not cleared by a store to the new real page by some other processor or mechanism. This clearing of reservations is unnecessary on processors that support the Store Conditional Page Mobility category.

The Store Conditional Page Mobility category does not provide a mechanism for the \textit{Store Conditional} instruction to detect that a virtual page has been moved to a new real page and back again to the original real page that was accessed by a \textit{Load and Reserve} instruction. Privileged software that moves a virtual page could clear the reservation on the processor it is running on in order to ensure that a \textit{Store Conditional} instruction executed by that processor does not succeed in this case. (The stores that occur naturally as part of moving the virtual page will cause any reservations, held by other processors, in the target real page to be lost.)

\subsection{Reservations}

The ability to emulate an atomic operation using \texttt{lwarx} and \texttt{stwcx}. is based on the conditional behavior of \texttt{stwcx}. the reservation created by \texttt{lwarx}, and the clearing of that reservation if the target storage location is modified by another processor or mechanism before the \texttt{stwcx}. performs its store.

A reservation is held on an aligned unit of real storage called a reservation granule. The size of the reservation granule is $2^n$ bytes, where $n$ is implementation-dependent but is always at least 4 (thus the minimum reservation granule size is a quadword) and, if the Store Conditional Page Mobility category is supported, where $2^n$ is not larger than the smallest real page size supported by the implementation. The reservation granule associated with effective address $EA$ contains the real address to which $EA$ maps. (*real_addr($EA$) in the RTL for the \textit{Load And Reserve} and \textit{Store Conditional} instructions stands for “real address to which $EA$ maps”.) The reservation also has an associated length, which is equal to the storage operand length, in bytes, of the \textit{Load And Reserve} instruction that established the reservation.

A processor has at most one reservation at any time. A reservation is established by executing a \texttt{lbarx}, \texttt{lharx}, \texttt{lwarx}, \texttt{idarx}, or \texttt{iqarx} instruction, as described in item 1 below, and is lost or may be lost, depending on the item, if any of the following occur. Items 1-8 apply only if the relevant access is performed. (For example, an access that would ordinarily be caused by an instruc-
tion might not be performed if the instruction causes the system error handler to be invoked.)

1. The processor holding the reservation executes another `lbarx`, `lharx`, `lwarx`, or `ldarx`: this clears the first reservation and establishes a new one.

2. The processor holding the reservation executes any `stbcx`, `sthcx`, `stwcx`, `stdcx`, or `stqcx`, regardless of whether the specified address matches the address specified by the `lbarx`, `lharx`, `lwarx`, `ldarx`, or `lqarx` that established the reservation, and regardless of whether the storage operand lengths of the two instructions are the same.

3. <TM> Any of the following occurs on the processor holding the reservation.
   a. The transaction state changes (from Non-transactional, Transactional, or Suspended state to one of the other two states; see Section 5.2, "Transactional Memory Facility States"), except in the following cases
      ■ If the change is from Transactional state to Suspended state, the reservation is not lost.
      ■ If the change is from Suspended state to Transactional state, the reservation is not lost if it was established in Transactional state.
      ■ If the change is caused by a `treclaim`, or `trechkpt`, instruction, whether the reservation is lost is undefined.
   b. The transaction nesting depth (see Section 5.4, "Transactional Memory Facility Registers") changes; whether the reservation is lost is undefined. (This item applies only if the processor is in Transactional state both before and after the change.)
   c. The processor is in Suspended state and executes a `Store Conditional` instruction (`stbcx`, `sthcx`, `stwcx`, `stdcx`, or `stqcx`) or a `waitrsv` instruction; the reservation is lost if it was established in Transactional state. In this case the `Store Conditional` instruction's store is not performed, and the `waitrsv` does not wait. (For `Store Conditional`, the reservation is also lost if it was established in Suspended state; see item 2.)

4. Some other processor executes a `Store`, `dcbz`, or `dcbzep`<E> that specifies a location in the same reservation granule.

5. Some other processor executes a `dcbtst`, `dcbtstep`<E>, or `dcbtstls`<E> that specifies a location in the same reservation granule: whether the reservation is lost is undefined. (For a `dcbtst` instruction that specifies a data stream, "location" in the preceding sentence includes all locations in the data stream.)

6. <E> Some other processor executes a `dcbx` that specifies a location in the same reservation granule: the reservation is lost if the instruction causes the target block to be newly established in a data cache or to be modified; otherwise whether the reservation is lost is undefined.

7. <E> Some other processor executes a `dcbi` that specifies a location in the same reservation granule: the reservation may be lost if the instruction is treated as a `Store`.

8. <S> Any processor modifies a Reference or Change bit (see Book III-S) in the same reservation granule: whether the reservation is lost is undefined.

9. Some mechanism other than a processor modifies a storage location in the same reservation granule.

10. An interrupt (see Book III) occurs on the processor holding the reservation: for the Embedded environment, the reservation may be lost if the interrupt is asynchronous. (For the Server environment the reservation is not lost. However, for both environments, system software invoked by interrupts may clear the reservation.)

11. Implementation-specific characteristics of the coherence mechanism cause the reservation to be lost.

---

Virtualized Implementation Note

A reservation may be lost if:

- Software executes a privileged instruction or utilizes a privileged facility
- Software accesses storage not intended for general-purpose programming
- Software executes a Decorated Storage instruction <DS>
- Software accesses a Device Control Register
1.7.3.2 Forward Progress

Forward progress in loops that use \texttt{lwarx} and \texttt{stwcx} is achieved by a cooperative effort among hardware, system software, and application software. The architecture guarantees that when a processor executes a \texttt{lwarx} to obtain a reservation for location X and then a \texttt{stwcx} to store a value to location X, either

1. the \texttt{stwcx} succeeds and the value is written to location X, or
2. the \texttt{stwcx} fails because some other processor or mechanism modified location X, or
3. the \texttt{stwcx} fails because the processor’s reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor or mechanism writing elsewhere in the reservation granule, cancellation caused by the operating system in managing certain limited resources such as real storage, and cancellation caused by any of the other effects listed in see Section 1.7.3.1.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.
## 1.8 Transactions [Category: Transactional Memory]

A transaction is a group of instructions that collectively have unique storage access behavior intended to facilitate parallel programming. (It is possible to nest transactions within one another. The description in this chapter will ignore nesting because it does not have a significant impact on the properties of the memory model. Nesting and its consequences will be described elsewhere.) Sequences of instructions that are part of the transaction may be interleaved with sequences of Suspended state instructions that are not part of the transaction. A transaction is said to “succeed” or to “fail,” and failure may happen before all of the instructions in the transaction have completed. If the transaction fails, it is as if the instructions that are part of the transaction were never executed. If the transaction succeeds, it appears to execute as an atomic unit as viewed by other processors and mechanisms.

(Although the transaction appears to execute atomically, some knowledge of the inner workings will be necessary to avoid apparent paradoxes in the rest of the model. These details are described below.) The execution of Suspended state sequences have the same effect that the sequence would have in the absence of a transaction, independent of the success or failure of the transaction, including accessing storage according to the weakly consistent storage model or SAO, based on storage attributes. Upon failure, normal execution continues at the failure handler. Except for the rollback of the effects of transactional instructions upon transaction failure, as viewed by the executing thread, the interleaved sequences of Transactional and Suspended state instructions appear to execute according to the sequential execution model. See Chapter 5. “Transactional Memory Facility [Category: Transactional Memory]” on page 795 for more details.

The unique attributes of the storage model for transactions are described below.

Transaction processing does not support the rollback of operations on the reservation mechanism. To prevent this possibility, a reservation is lost as a result of a state change from Transactional to Non-transactional or Non-transactional to Transactional. It is possible to successfully complete an atomic update in Transactional state, though such a sequence would have no benefit. It is also possible to complete an atomic update in Suspended state, or straddling an interval in Suspended state if Suspended state is entered via an interrupt or tsuspend, and exited via tremuse, rfebb, rfid, hrfid, or mtmsrd. However, an atomic update will not succeed if only one of the Load and Reserve / Store Conditional instruction pair is executed in Suspended state.

### Programming Note

Note that if a Store Conditional instruction within a transaction does not store, it may still be possible for the transaction to succeed. Software must not depend on the two operations having the same outcome. For example, software must not use success of an enclosing transaction as a replacement for checking the condition code from a transactional Store Conditional instruction.

### Virtualized Implementation Note

On a virtualized implementation, Case 3 includes reservation loss caused by the virtualization software. Thus, on a virtualized implementation, a reservation may be lost at any time without apparent cause. The virtualization software participates in any forward progress assurances, as described above.

### Programming Note

The architecture does not include a “fairness guarantee”. In competing for a reservation, two processors can indefinitely lock out a third.

### Programming Note

Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparent storage paradoxes. Consider, for example, a case where variable X has initial value zero, is updated transactionally to one, is read in Suspended state, subsequently the transaction fails, and variable X is read again. In the absence of external conflicts, the observed sequence of values will be zero, one, zero: old, new, old.

Performing an atomic update on X in Suspended state may be even more confusing. Suppose the atomic sequence increments X, but that the only way to have X=1 is via the transactional store that occurs before entering Suspended state. The store conditional, if it succeeds, will store X=2 and in so doing, kill the transaction. But with the transaction having failed, X was never equal to one.

The flexibility of the Suspended state programming model can create unintuitive results. It must be used with care.

Successful transactions are serialized in some order, and no processor or mechanism is able to observe the accesses caused by any subset of these transactions as occurring in an order that conflicts with this order. Specifically, let processor i execute transactions 0, 1, ..., j, j+1, ..., where only successful transactions are numbered, and the numbering reflects program order. Let T_ij be transaction j on processor i. Then there is an ordering of the T_ij such that no processor or mechanism is able to observe the accesses caused by the transactions T_ij in an order that conflicts with this order.
ing. Note that Suspended state storage accesses are not included in the serialization property.

--- Programming Note ---

The ordering of the $T_{ij}$ for a given $i$ is consistent with program order for processor $i$.

Because of the difference between a transaction's instantaneous appearance and the finite time required to execute it in an implementation, it is exposed to changes in memory management state in a way that is not true for individual accesses. A change to the translation or protection state that would prevent any access from taking place at any time during its processing for the transaction compromises the integrity of the transaction. Any such change must either be prevented or must cause the transaction to fail. The architecture will automatically fail a transaction if the memory management state change is accomplished using $tlib$. An implementation may overdetect such conflicts between the $tlib$ and the transaction footprint. (Overdetection may result from the technique used to detect the conflict. A bloom filter may be used, as an example. Subsequent references to translation invalidation conflicts implicitly include any cases of spurious overdetection.) Changes made in some other manner must be managed by software, for example by explicitly terminating any affected transactions. Examples of instructions that require software management are $tlib$, $slbie$, $slbia$, and $tliba$.

The atomic nature of a transaction, together with the cumulative memory barrier created by the transaction and the memory barriers created by $tbegin$ and $tend$, described below, has the potential to eliminate the need for explicit memory barriers within the transaction, and before and after the transaction as well. However, since there may be a desire to preserve existing algorithms while exploiting transactions, the interaction of memory barriers and transactions is defined. In the presence of transactions, storage access ordering is the same as if no transactions are present, with the following exceptions. Memory barriers that are created while the transaction is running (other than the integrated cumulative memory barrier of the transaction described below), data dependencies, and SAO do not order transactional stores. Instead, transactional stores are grouped together into an "aggregate store," which is performed as an atomic unit with respect to other processors and mechanisms when the transaction succeeds, after all the transactional loads have been performed. With this store behavior, the appearance of transactional atomicity is created in a manner similarly to that for a Load and Reserve / Store Conditional pair. Success of the transaction is conditional on the storage locations specified by the loads not having been stored into by a more recent Suspended state store or by any store by another processor or mechanism since the load was performed. (There are additional conditions for the success of transactions.)

A $tbegin$, instruction that begins a successful transaction creates a memory barrier that immediately precedes the transaction and orders storage accesses pairwise, as follows. Let $A$ and $B$ be sets of storage accesses as defined below. For each pair $a_i b_j$ of storage accesses such that $a_i$ is in $A$ and $b_j$ is in $B$, the memory barrier ensures that $a_i$ will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before $b_j$ is performed with respect to that processor or mechanism. Set $A$ contains all data accesses caused by instructions preceding the $tbegin$, that are neither Write Through Required nor Caching Inhibited. Set $B$ contains all data accesses caused by instructions following the $tbegin$, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited.

--- Programming Note ---

The reason the creation of the memory barrier by $tbegin$, is specified to be contingent on the transaction succeeding is that delaying the creation may improve performance, and does not seriously inconvenience software.

A successful transaction has an integrated memory barrier behavior. When a processor (P1) executes a $tend$, instruction and $tend$, processing determines that the transaction will succeed, a memory barrier is created, which orders storage accesses pairwise, as follows. Let $A$ and $B$ be sets of storage accesses as defined below. For each pair $a_i b_j$ of storage accesses such that $a_i$ is in $A$ and $b_j$ is in $B$, the memory barrier ensures that $a_i$ will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before $b_j$ is performed with respect to that processor or mechanism. Set $A$ contains all non-transactional data accesses by other processors and mechanisms that have been performed with respect to P1 before the memory barrier is created and are neither Write Through Required nor Caching Inhibited. Set $B$ contains the aggregate store and all non-transactional data accesses by other processors and mechanisms that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in set $B$. Note that the cumulative memory barrier does not order Suspended state storage accesses interleaved with the transaction.

A $tend$, instruction that ends a successful transaction creates a memory barrier that immediately follows the transaction and orders storage accesses pairwise, as follows. Let $A$ and $B$ be sets of storage accesses as defined below. For each pair $a_i b_j$ of storage accesses such that $a_i$ is in $A$ and $b_j$ is in $B$, the memory barrier ensures that $a_i$ will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before $b_j$ is performed with respect to that processor or mechanism.
mechanism. Set A contains all data accesses caused by instructions preceding the `tend`, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the `tend` that are neither Write Through Required nor Caching Inhibited.

**Programming Note**

The barriers that are created by the execution of a successful transaction (those associated with `tbegin`, `tend`, and the integrated cumulative barrier) render most explicit barriers in and around transactions redundant. An exception is when there is a need to establish order among Suspended state accesses.

### 1.8.1 Rollback-Only Transactions

A Rollback-Only Transaction (ROT) is a sequence of instructions that is executed, or not, as a unit. The purpose of the ROT is to enable bulk speculation of instructions with minimum overhead. It leverages the rollback mechanism that is invoked as part of transaction failure handling, but has reduced overhead in that it does not have the full atomic nature of the transaction and its synchronization and serialization properties. The absence of a (normal) transaction's atomic quality means that a ROT must not be used to manipulate shared data.

More specifically, a ROT differs from a normal transaction as follows.

- ROTs are not serialized.
- There are no memory barriers created by `tbegin` and `tend`.
- A ROT has no integrated cumulative memory barrier.
- There is no monitoring of storage locations specified by loads for modification by other processors and mechanisms between the performing of the loads and the completion of the ROT.
- The stores that are included in the ROT need not appear to be performed as an aggregate store. (Implementations are likely to provide an aggregate store appearance, but the correctness of the program must not depend on the aggregate store appearance.)

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III-S) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the “address translation mechanism” or the “storage protection mechanism”; see Book III).

**Programming Note**

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Section 1.6.2 and Section 1.6.4 of this Book and in Book III.

For each instance of executing an instruction from location X, the instruction may be fetched multiple times.

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the result of attempting to execute the instruction is boundedly undefined except as described in Section 1.9.1, “Concurrent Modification and Execution of Instructions” on page 752.

### 1.9 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.
Programming Note

Following are examples of how to make instruction storage consistent with data storage. Because the optimal instruction sequence to make instruction storage consistent with data storage may vary between systems, many operating systems will provide a system service to perform this function.

**Case 1:** The given program does not modify instructions executed by another program nor does another program modify the instructions executed by the given program.

Assume that location X previously contained the instruction A0; the program modified one of more bytes of that location such that, in data storage, the location contains the instruction A1; and location X is wholly contained in a single cache block. The following instruction sequence will make instruction storage consistent with data storage such that if the *isync* was in location X-4, the instruction A1 in location X would be executed immediately after the *isync*.

```plaintext
dcbst X  #copy the block to main storage sync    #order copy before invalidation icbi X  #invalidate copy in instr cache sync    #order invalidation before store # to flag stw r0,flag  #set flag indicating instruction # storage is now consistent
```

**Case 2:** One or more programs execute the instructions that are concurrently being modified by another program.

Assume program A has modified the instruction at location X and other programs are waiting for program A to signal that the new instruction is ready to execute. The following instruction sequence will make instruction storage consistent with data storage and then set a flag to indicate to the waiting programs that the new instruction can be executed.

```plaintext
li r0,1  #put a 1 value in r0 dcbst X  #copy the block in main storage sync    #order copy before invalidation icbi X  #invalidate copy in instr cache sync    #order invalidation before store # to flag stw r0,flag  #set flag indicating instruction # storage is now consistent
```

The following instruction sequence, executed by the waiting program, will prevent the waiting programs from executing the instruction at location X until location X in instruction storage is consistent with data storage, and then will cause any prefetched instructions to be discarded.

```plaintext
lwz r0,flag  #loop until flag = 1 (when 1 is cmpwi r0,1 # loaded, location X in inst’n bne $-8 # storage is consistent with # location X in data storage) isync    #discard any prefetched inst’ns
```

In the preceding instruction sequence any context synchronizing instruction (e.g., *rfid*) can be used instead of *isync*. (For Case 1 only *isync* can be used.)

For both cases, if two or more instructions in separate data cache blocks have been modified, the *dcbst* instruction in the examples must be replaced by a sequence of *dcbst* instructions such that each block containing the modified instructions is copied back to main storage. Similarly, for *icbi* the sequence must invalidate each instruction cache block containing a location of an instruction that was modified. The *sync* instruction that appears above between "*dcbst* X" and "*icbi* X" would be placed between the sequence of *dcbst* instructions and the sequence of *icbi* instructions.
1.9.1 Concurrent Modification and Execution of Instructions

The phrase “concurrent modification and execution of instructions” (CMODX) refers to the case in which a processor fetches and executes an instruction from instruction storage which is not consistent with data storage or which becomes inconsistent with data storage prior to the completion of its processing. This section describes the only case in which executing this instruction under these conditions produces defined results.

In the remainder of this section the following terminology is used.

- Location $X$ is an arbitrary word-aligned storage location.
- $X_0$ is the value of the contents of location $X$ for which software has made the location $X$ in instruction storage consistent with data storage.
- $X_1, X_2, ..., X_n$ are the sequence of the first $n$ values occupying location $X$ after $X_0$.
- $X_n$ is the first value of $X$ subsequent to $X_0$ for which software has again made instruction storage consistent with data storage.
- The “patch class” of instructions consists of the I-form Branch instruction ($b[l][a]$) and the preferred no-op instruction ($ori0,0,0$).

If the instruction from location $X$ is executed after the copy of location $X$ in instruction storage is made consistent for the value $X_0$ and before it is made consistent for the value $X_n$, the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values $X_1, ..., X_n$ into location $X$ are atomic stores that modify all four bytes of location $X$.
2. Each $X_i, 0 \leq i \leq n$, is a patch class instruction.
3. Location $X$ is in storage that is Memory Coherence Required.

If these conditions are satisfied, the result of each execution of an instruction from location $X$ will be the execution of some $X_i, 0 \leq i \leq n$. The value of the ordinate $i$ associated with each value executed may be different and the sequence of ordinates $i$ associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction at location $X$ could be the sequence $X_i, X_{i+2}, X_{i+1}$). If these conditions are not satisfied, the results of each such execution of an instruction from location $X$ are boundedly undefined, and may include causing inconsistent information to be presented to the system error handler.

---

**Programming Note**

An example of how failure to satisfy the requirements given above can cause inconsistent information to be presented to the system error handler is as follows. If the value $X_0$ (an illegal instruction) is executed, causing the system illegal instruction handler to be invoked, and before the error handler can load $X_0$ into a register, $X_0$ is replaced with $X_1$, an Add Immediate instruction, it will appear that a legal instruction caused an illegal instruction exception.

---

**Programming Note**

It is possible to apply a patch or to instrument a given program without the need to suspend or halt the program. This can be accomplished by modifying the example shown in the Programming Note at the end of Section 1.9 where one program is creating instructions to be executed by one or more other programs.

In place of the Store to a flag to indicate to the other programs that the code is ready to be executed, the program that is applying the patch would replace a patch class instruction in the original program with a Branch instruction that would cause any program executing the Branch to branch to the newly created code. The first instruction in the newly created code must be an **isync**, which will cause any prefetched instructions to be discarded, ensuring that the execution is consistent with the newly created code. The instruction storage location containing the **isync** instruction in the patch area must be consistent with data storage with respect to the processor that will execute the patched code before the Store which stores the new Branch instruction is performed.

---

**Programming Note**

It is believed that all processors that comply with versions of the architecture that precede Version 2.01 support concurrent modification and execution of instructions as described in this section if the requirements given above are satisfied, and that most such processors yield boundedly undefined results if the requirements given above are not satisfied. However, in general such support has not been verified by processor testing. Also, one such processor is known to yield undefined results in certain cases if the requirements given above are not satisfied.
Chapter 2. Effect of Operand Placement on Performance

The placement (location and alignment) of operands in storage affects relative performance of storage accesses, and may affect it significantly. The best performance is guaranteed if storage operands are aligned to the natural boundary of the data accessed. Performance of storage accesses varies depending on the following:

1. Operand Size
2. Operand Alignment
3. Crossing no boundary
4. Crossing a cache block boundary
5. Crossing a virtual page boundary

Figure 1 on page 754 specifies which storage accesses that are permitted, but not necessarily required, to invoke the system alignment error handler, resulting in significantly degraded performance.

Specific platforms may set more restrictive requirements on which storage accesses an implementation is allowed to invoke the system alignment error handler.

The Move Assist instructions have no alignment requirements.
### Figure 1. Unaligned storage accesses

<table>
<thead>
<tr>
<th>Class</th>
<th>Size</th>
<th>Byte Align&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Access may cause invocation of the system alignment error handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-byte</td>
<td>any</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>2-byte</td>
<td>2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 2</td>
<td>Yes&lt;sup&gt;3&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>4-byte</td>
<td>4</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 4</td>
<td>Yes&lt;sup&gt;2&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>8</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 8</td>
<td>Yes&lt;sup&gt;2&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 16</td>
<td>Yes&lt;sup&gt;5&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td>4-byte</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 4</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>odd 4</td>
<td>Yes&lt;sup&gt;2&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 4</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 16</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>VSX</td>
<td>4-byte</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 4</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>odd 4</td>
<td>Yes&lt;sup&gt;2&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 4</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>odd 8 or odd 4</td>
<td>Yes&lt;sup&gt;2&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 4</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>any</td>
<td>any</td>
<td>No</td>
</tr>
</tbody>
</table>

1. "N" means any N-byte alignment
   "odd N" means any N-byte alignment but not 2×N-byte alignment
   "< N" means any non-N-byte alignment
2. Including Move Assist instructions (*lswi*, *lswx*, *stswi*, and *stswx*).
3. For Category: Server, may be invoked when the storage access uses Little-Endian byte ordering or when the storage access spans a virtual page boundary.
   For Category: Embedded, may be invoked for any unaligned access.
4. Vector storage access instructions are always quadword-aligned.
5. May be invoked for any unaligned access.
2.1 Instruction Restart

In this section, “Load instruction” includes the Cache Management and other instructions that are stated in the instruction descriptions to be “treated as a Load”, and similarly for “Store instruction”.

The following instructions are never restarted after having accessed any portion of the storage operand (unless the instruction causes a “Data Address Watchpoint match”, for which the corresponding rules are given in Book III).

1. A Store instruction that causes an atomic access and, for the Embedded environment, accesses storage that is Guarded
2. A Load instruction that causes an atomic access to storage that is Guarded and, for the Server environment, is also Caching Inhibited.

Any other Load or Store instruction may be partially executed and then aborted after having accessed a portion of the storage operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed. Additional restrictions on the partial execution of instructions are described in Section 6.6 of Book III-S and Section 7.7 of Book III-E.

--- Programming Note ---

In order to ensure that the contents of registers are preserved to the extent that a partially executed instruction can be re-executed correctly, the registers that are preserved must satisfy the following conditions. For any given instruction, zero or more of the conditions applies.

- For a fixed-point Load instruction that is not a multiple or string form, or for an eciw instruction, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an update form Load or Store instruction, the contents of register RA are not altered.

--- Programming Note ---

There are many events that might cause a Load or Store instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however, the contents of some registers may have been altered and some bytes within the storage operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

1. Load Multiple, Load String: Some registers in the range of registers to be loaded may have been altered.
2. Any Store instruction, dcbz: Some bytes of the storage operand may have been altered.
Chapter 3. Management of Shared Resources

The facilities described in this section provide the means to control the use of resources that are shared with other processors.

3.1 Program Priority Registers

The Program Priority Register (PPR) is a 64-bit register that controls the program’s priority. The PPR provides access to the full 64-bit PPR, and the Program Priority Register 32-bit (PPR32) provides access to the upper 32 bits of the PPR. The Embedded environment only provides access to PPR32. The layouts of the PPR and PPR32 are shown in Figure 2.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:13</td>
<td>Program Priority (PRI) (PPR32:43:45)</td>
</tr>
<tr>
<td>001</td>
<td>very low</td>
</tr>
<tr>
<td>010</td>
<td>low</td>
</tr>
<tr>
<td>011</td>
<td>medium low</td>
</tr>
<tr>
<td>100</td>
<td>medium</td>
</tr>
<tr>
<td>101</td>
<td>medium high</td>
</tr>
</tbody>
</table>

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 4.3.6.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

If other values are written to this field, the PRI field is not changed. (See Section 4.3.5 of Book III-S for additional information.)

All other fields are reserved.

Figure 2. Program Priority Register

Programming Note

The ability to access the low-order half of the PPR (and thus the use of mfppr and mtppr) might be phased out in a future version of the architecture.

Programming Note

By setting the PRI field, a programmer may be able to improve system throughput by causing system resources to be used more efficiently.

E.g., if a program is waiting on a lock (see Section B.2), it could set low priority, with the result that more processor resources would be diverted to the program that holds the lock. This diversion of resources may enable the lock-holding program to complete the operation under the lock more quickly, and then relinquish the lock to the waiting program.

Programming Note

or Rx,Rx,Rx can be used to modify the PRI field; see Section 3.2.

Programming Note

When the system error handler is invoked, the PRI field may be set to an undefined value.

3.2 “or” Instruction

Setting the PPR

The or Rx,Rx,Rx (see Book I) instruction can be used to set PPRPRI as shown in Figure . or. Rx,Rx,Rx does not set PPRPRI.

<table>
<thead>
<tr>
<th>Rx</th>
<th>PPRPRI</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>001</td>
<td>very low</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>low</td>
</tr>
<tr>
<td>6</td>
<td>011</td>
<td>medium low</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>medium</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>medium high</td>
</tr>
</tbody>
</table>
Priority levels for or Rx,Rx,Rx

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 4.3.6 of Book III-S.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

The following forms of or Rx,Rx,Rx provide hints about usage of shared processor resources.

“or” Shared Resource Hints

or 27,27,27

This form of or provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released for use by other processors.

or 29,29,29

This form of or provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released until all outstanding storage accesses to caching-inhibited storage have been completed.

or 30,30,30

This form of or provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released until all outstanding storage accesses to cacheable storage for which the data is not in the cache have been completed.

Extended Mnemonics:

Additional extended mnemonics for the or hints:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>yield</td>
<td>or 27,27,27</td>
</tr>
<tr>
<td>mdoio</td>
<td>or 29,29,29</td>
</tr>
<tr>
<td>mdoom</td>
<td>or 30,30,30</td>
</tr>
</tbody>
</table>

**Programming Note**

**Warning:** Other forms of or Rx,Rx,Rx that are not described in this section and in Section 4.3.3, “or” Instruction”, in Book II may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (ori 0,0,0) should be used.
Chapter 4. Storage Control Instructions

4.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. The virtual page sizes
2. Coherence block size
3. Reservation granule size
4. An indication of the cache model implemented (e.g., Harvard-style cache, combined cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache block size
8. Data cache block size
9. Instruction cache associativity
10. Data cache associativity
11. Number of stream IDs supported for the stream variant of *dcbt*
12. Factors for converting the Time Base to seconds
13. Maximum transaction level

If the caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

4.2 Data Stream Control Register (DSCR) [Category: Stream]

The layout of the Data Stream Control Register (DSCR) is shown in Figure 3 below.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>Software Transient Enable (SWTE)</td>
<td>0   Must be 1 to apply the transient attribute to software-defined streams.</td>
</tr>
<tr>
<td>40</td>
<td>Hardware Transient Enable (HWTE)</td>
<td>0   HWTE is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Applies the transient attribute to hardware-detected streams.</td>
</tr>
<tr>
<td>41</td>
<td>Store Transient Enable (STE)</td>
<td>0   STE is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Applies the transient attribute to store streams.</td>
</tr>
<tr>
<td>42</td>
<td>Load Transient Enable (LTE)</td>
<td>0   LTE is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Applies the transient attribute to load streams.</td>
</tr>
<tr>
<td>43</td>
<td>Software Unit count Enable (SWUE)</td>
<td>0   SWUE is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Applies the unit count to software-defined streams.</td>
</tr>
<tr>
<td>44</td>
<td>Hardware Unit count Enable (HWUE)</td>
<td>0   HWUE is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Applies the unit count to hardware-detected streams.</td>
</tr>
<tr>
<td>45:54</td>
<td>Unit Count (UNITCNT)</td>
<td>Number of units in data stream.</td>
</tr>
<tr>
<td>55:57</td>
<td>Depth Attainment Urgency (URG)</td>
<td>This field indicates how quickly the prefetch depth should be reached for hardware-detected streams. Values and their meanings are as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0   default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   not urgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2   least urgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3   less urgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4   medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5   urgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6   more urgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7   most urgent</td>
</tr>
<tr>
<td>58</td>
<td>Load Stream Disable (LSD)</td>
<td>0   No effect.</td>
</tr>
</tbody>
</table>

Figure 3. Data Stream Control Register
1 Disables hardware detection and initiation of load streams.

59 **Stride-N Stream Enable (SNSE)**

0 No effect.
1 Enables the hardware detection and initiation of load and store streams that have a stride greater than a single cache block. Such load streams are detected only when LSD is also zero. Such store streams are detected only when SSE is also one.

61:63 **Default Prefetch Depth (DPFD)**

This field supplies a prefetch depth for hardware-detected streams and for software-defined streams for which a depth of zero is specified or for which `dcbt/dcbtst` with `TH=1010` is not used in their description. Values and their meanings are as follows.

- 0 default (LPCRDPFD)
- 1 none
- 2 shallowest
- 3 shallow
- 4 medium
- 5 deep
- 6 deeper
- 7 deepest

The contents of the DSCR affect how a processor handles hardware-detected and software-defined data streams. The DSCR provides the only means by which software can control or supply information for hardware-detected data streams. The DPFD, UNITCNT, and transient fields may also be used instead of the `TH=01010` variant of `dcbt` for software-defined data streams, especially when multiple streams have these attributes in common. See Section 4.3.2, “Data Cache Instructions” on page 763, for information on streams and how software may specify them.

**Programming Note**

The purpose of Depth Attainment Urgency is to regulate the rate of prefetch generation from the cycle at which the hardware first detects an incipient stream until the cycle when the prefetch Depth is reached. A more urgent setting will benefit applications that are dominated by short to medium length streams, because otherwise prefetching does not occur rapidly enough to benefit them. In contrast, applications that frequently cause unproductive prefetches due to stream mispredicts will benefit from a less urgent setting.

Unlike the Depth, the Depth Attainment Urgency applies only to hardware-detected streams. Furthermore, the DSCR provides the only point of control for this parameter. Software-defined streams are assumed not to have the correctness risk associated with hardware streams, and therefore are set to reach their depth relatively quickly.

**Programming Note**

In versions of the architecture that precede Version 2.07, `mspr` specifying the DSCR caused all active and nascent data streams to cease to exist. In those versions of the architecture, the DSCR was used as an overall control mechanism to specify a single global profile for all streams. Beginning with Version 2.07, the DSCR is intended to control and accelerate the creation of new streams without disturbing existing streams.

**Programming Note**

The URG, LSD, SNSE and SSE fields do not affect the initiation of streams specified using the `dcbt` and `dcbtst` instructions.

Note that even when SNSE is not set, hardware may detect Stride-N streams in intervals when they access elements that map to sequential cache blocks.

**Programming Note**

In order for the DSCR to apply the transient attribute to streams, at least two of the four enable bits must be set: one to choose a type of access (load or store), and one to choose a kind of prefetching (software-defined or hardware-detected).
4.3 Cache Management Instructions

The *Cache Management* instructions obey the sequential execution model except as described in Section 4.3.1.

In the instruction descriptions the statements “this instruction is treated as a *Load*” and “this instruction is treated as a *Store*” mean that the instruction is treated as a *Load* (*Store*) from (to) the addressed byte with respect to address translation, the definition of program order on page 735, storage protection, reference and change recording (*S*), and the storage access ordering described in Section 1.7.1 and is treated as a *Read* (*Write*) from (to) the addressed byte with respect to debug events unless otherwise specified. (See Book III-E.)

**Programming Note**

Accesses that are caused by or associated with *Cache Management* instructions that are “treated as a *Load*” or “treated as a *Store*” are not subject to the special ordering rules described for SAO storage. These accesses are always performed in accordance with the weakly consistent storage model.

Some *Cache Management* instructions contain a CT field that is used to specify a cache level within a cache hierarchy or a portion of a cache structure to which the instruction is to be applied. The correspondence between the CT value specified and the cache level is shown below.

<table>
<thead>
<tr>
<th>CT Field Value</th>
<th>Cache Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Primary Cache</td>
</tr>
<tr>
<td>2</td>
<td>Secondary Cache</td>
</tr>
</tbody>
</table>

CT values not shown above may be used to specify implementation-dependent cache levels or implementation-dependent portions of a cache structure.
### 4.3.1 Instruction Cache Instructions

#### Instruction Cache Block Invalidate X-form

**icbi RA, RB**

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the instruction cache of this processor, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a **Load** (see Section 4.3), except that reference and change recording need not be done.

**Special Registers Altered:**

None

---

#### Instruction Cache Block Touch X-form

**icbt CT, RA, RB**

Let the effective address (EA) be the sum (RA|0)+(RB).

The **icbt** instruction provides a hint that the program will probably soon execute code from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

The hint is ignored if the block is Caching Inhibited.

This instruction treated as a **Load** (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

**Special Registers Altered:**

None

---

**Programming Note**

Because the instruction is treated as a **Load**, the effective address is translated using translation resources that are used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches (see Book III).
4.3.2 Data Cache Instructions

The Data Cache instructions control various aspects of the data cache.

**TH field in the dcbt and dcbtst instructions**

Described below are the TH field values for the `dcbt` and `dcbtst` instructions. For all TH field values which are not listed, the hint provided by the instruction is undefined.

**TH=0b00000**

If TH=0b00000, the `dcbt/dcbtst` instruction provides a hint that the program will probably soon access the block containing the byte addressed by EA.

**TH=0b00000 - 0b00111 [Category: Cache Specification]**

In addition to the hint specified above for the TH field value of 0b00000, an additional hint is provided indicating that placement of the block in the cache specified by the TH field might also improve performance. The correspondence between each value of the TH field and the cache to be specified is the same as the correspondence between each value the CT field and the cache to be specified as defined in Section 4.3. The hints corresponding to values of the TH field not supported by the implementation are undefined.

**TH=0b01000 - 0b01111 [Category: Stream]**

The `dcbt/dcbtst` instructions provide hints regarding a sequence of accesses to data elements, or indicate the expected use thereof. Such a sequence is called a “data stream”, and a `dcbt/dcbtst` instruction in which TH is set to one of these values is said to be a “data stream variant” of `dcbt/dcbtst`. In the remainder of this section, “data stream” may be abbreviated to “stream”.

A data stream to which a program may perform Load accesses is said to be a “load data stream”, and is described using the data stream variants of the `dcbt` instruction. A data stream to which a program may perform Store accesses is said to be a “store data stream”, and is described using the data stream variants of the `dcbtst` instruction.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

Each data element is associated with a unit of storage, which is the aligned 128-byte location in storage that contains the first byte of the element. The data stream variants may be used to specify the address of the beginning of the data stream, the displacement (stride) between the first byte of successive elements, and the number of unique units of storage that are associated with all of the data elements. If the stride is specified, both the stride and the address of the first element are specified at 4 byte granularity. If the stride is not specified, the address of the first element is the address of the first unit.

**Programming Note**

The architecture does not provide a way to specify the size of the data elements that compose a stream. An implementation may assume some fixed size for all data elements. As a result, depending on the offset, stride, and size (and in particular whether the elements are aligned), the implementation may reduce the latency for accessing only a portion of some of the elements. A future version of the architecture may enable the specification of element size to avoid this limitation.

**Bit(s) Description**

0:56 **EATRUNC**

High-order 57 bits of the effective address of the first element of the data stream. (i.e., the effective address of the first unit of the stream is `EATRUNC` | 70)

57 **Direction** (D)

0 Subsequent elements have increasing addresses.

1 Subsequent elements have decreasing addresses.
58  Unlimited/GO (UG)
    0  No information is provided by the UG field.
    1  The number of elements in the data stream is unlimited, the elements are adjacent to each other, the program's need for each element of the stream is not likely to be transient, and the program will probably soon access the stream.

59  Reserved
60:63  Stream ID (ID)
    Stream ID to use for this data stream.

01010  The dcbt/dcbtst instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon access data streams that have been described using data stream variants of the dcbt/dcbtst instruction, or will probably no longer access such data streams.

The EA is interpreted as follows. If GO=1 and S≠0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.

<table>
<thead>
<tr>
<th></th>
<th>GO</th>
<th>S</th>
<th>DEP</th>
<th>UNITCNT</th>
<th>T</th>
<th>U</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>35</td>
<td>36</td>
<td>39</td>
<td>47</td>
<td>57</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit(s) Description

0:31  GO

32 0  No information is provided by the GO field.
   1  For dcbt, the program will probably soon access all nascent load and store data streams that have been completely described, and will probably no longer access all other nascent load and store data streams. All other fields of the EA are ignored. ("Nascent" and "completely described" are defined below.) For dcbtst, this field value holds no meaning and is treated as though it were zero.

33:34  Stop (S)

00  No information is provided by the S field.
   01  Reserved
   10  The program will probably no longer access the data stream (if any) associated with the specified stream ID. (All other fields of the EA except the ID field are ignored.)

11  For dcbt, the program will probably no longer access the load and store data streams associated with all stream IDs. (All other fields of the EA are ignored.) For dcbtst, this field value holds no meaning, and is treated as though it were 0b00.

35  Reserved
36:38  Depth (DEP)
    The DEP field provides a relative estimate of how many elements ahead of the point of stream use the latency-reducing actions should go. This value reflects a comparison of the rate of consumption of the elements of the data stream and the latency to bring an arbitrary element of the stream into cache. The values are as follows.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>35</td>
<td>36</td>
<td>39</td>
<td>47</td>
<td>57</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

39:46  Reserved
47:56  UNITCNT
    Number of units in data stream.

57  Transient (T)
    If T=1, the program's need for each element of the data stream is likely to be transient (i.e., the time interval during which the program accesses the element is likely to be short).

58  Unlimited (U)
    If U=1, the number of units in the data stream is unlimited (and the UNITCNT field is ignored).

59  Reserved
60:63  Stream ID (ID)
    Stream ID to use for this data stream (GO=0 and S=0b00), or stream ID associated with the data stream which the program will probably no longer access (S=0b10).
Chapter 4. Storage Control Instructions

The `dcbt/dcbtst` instruction provides a hint that describes certain attributes of a data stream.

The EA is interpreted as follows.

<table>
<thead>
<tr>
<th>Bit(s) Description</th>
<th>Bit(s) Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31 Reserved</td>
<td>32:49 Stride</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>50 Reserved</td>
<td>51:55 Offset</td>
</tr>
<tr>
<td></td>
<td>60:63 Stream ID (ID)</td>
</tr>
</tbody>
</table>

The number of elements ahead of the point of stream use indicated by a given depth value may differ across implementations, as may the latency to bring a given element into the cache. To achieve optimum performance, some experimentation with different depth values may be necessary.

A program should use a `dcbt/dcbtst` instruction with TH=0b01011 only when the stride is larger than 128 bytes. Otherwise, consecutive units will be accessed, so the additional stream information has no benefit.

If the specified stream ID value is greater than m-1, where m is the number of stream IDs provided by the implementation, and either (a) TH=0b01000 or TH=0b01011, or (b) TH=0b01010 with GO=0 and S=0b11, no hint is provided by the instruction.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint(s) for the stream.

- A data stream for which only descriptive hints have been provided (by `dcbt/dcbtst` instructions with TH=0b01000 and UG=0, TH=0b01010 and GO=0 and S=0b00, and/or with TH=0b01011) is said to be "nascent". A nascent data stream for which all relevant descriptive hints have been provided (by the `dcbt/dcbtst` usages listed in the preceding sentence) is considered to be "completely described". The order of descriptive hints with respect to one another is unimportant.

- A data stream for which a hint has been provided (by a `dcbt/dcbtst` instruction with TH=0b01000 and UG=1 or `dcbt` with TH=0b01010 and S=0b00) that the program will probably soon access it is said to be "active".

- A data stream that is either nascent or active is considered to "exist".

- A data stream for which a hint has been provided (e.g., by a `dcbt` instruction with TH=0b01010 and S=0b00) that the program will probably no longer access it is considered no longer to exist.

The hint provided by a `dcbt/dcbtst` instruction with TH=0b01000 and UG=1 implicitly includes a hint that the program will probably no longer access the data stream (if any) previously associated with the specified stream ID. The hint provided by a `dcbt/dcbtst` instruction with TH=0b01000 and UG=0, or with TH=0b01010 and GO=0 and S=0b00, or with TH=0b01011 implicitly includes a hint that the program will probably no longer access the active data stream (if any) previously associated with the specified stream ID.

If a data stream is specified without using a `dcbt/dcbtst` instruction with TH=0b01010 and GO=0 and S=0b00, then the number of elements in the stream is unlimited, and the program's need for each element of the stream is not likely to be transient. If a data stream is specified without using a `dcbt/dcbtst` instruction with
TH=0b01011, then the stream will access consecutive units of storage.

Interrupts (see Book III) cause all existing data streams to cease to exist. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a page.
To obtain the best performance across the widest range of implementations that support the data stream variants of `dcbt/dcbtst`, the programmer should assume the following model when using those variants.

- The processor's response to a hint that the program will probably soon access a given data stream is to take actions that reduce the latency of accesses to the first few elements of the stream. (Such actions may include prefetching cache blocks into levels of the storage hierarchy that are “near” the processor.) Thereafter, as the program accesses each successive element of the stream, the processor takes latency-reducing actions for additional elements of the stream, pacing these actions with the program's accesses (i.e., taking the actions for only a limited number of elements ahead of the element that the program is currently accessing).

The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, is to stop taking latency-reducing actions for the stream.

- A data stream having finite length ceases to exist when the latency-reducing actions have been taken for all elements of the stream.

- If the program ceases to need a given data stream before having accessed all elements of the stream (always the case for streams having unlimited length), performance may be improved if the program then provides a hint that it will no longer access the stream (e.g., by executing the appropriate `dcbt` instruction with \( TH=0b01010 \) and \( S\neq0b00 \)).

- At each level of the storage hierarchy that is “near” the processor, elements of a data stream that is specified as transient are most likely to be replaced. As a result, it may be desirable to stagger addresses of streams (choose addresses that map to different cache congruence classes) to reduce the likelihood that an element of a transient stream will be replaced prior to being accessed by the program.

- Processors that comply with versions of the architecture that do not support the TH field at all treat \( TH=0b01000, 0b01010, \) and \( 0b01011 \) as if \( TH=0b00000 \).

- A single set of stream IDs is shared between the `dcbt` and `dcbtst` instructions.

- On some implementations, data streams that are not specified by software may be detected by the processor. Such data streams are called “hardware-detected data streams”. On some such implementations, data stream resources (resources that are used primarily to support data streams) are shared between software-specified data streams and hardware-detected data streams. On these latter implementations, the programming model includes the following.

  - Software-specified data streams take precedence over hardware-detected data streams in use of data stream resources.

  - The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, includes releasing the associated data stream resources, so that they can be used by hardware-detected data streams.
Programming Note

The latency-reducing actions taken in response to a program's hints about access to a data stream, including the depth and urgency parameters, may vary based on its behavior and on the behavior of other programs sharing platform resources, as well as on the design of the platform resources they use. Without actually changing the stream specification or DSCR parameters, the processor may adjust its actions (e.g. slow down prefetches or be more selective choosing them) based on their effectiveness and on the availability of storage bandwidth. In general, the goal of this variation is to improve overall system performance and fairness across the set of programs that share resources. There often will be a performance benefit, however, from adjusting stream specifications to the platform and co-resident programs to adjust for these actions by the processor.
This Programming Note describes several aspects of using the data stream variants of the `dcbt` and `dcbtst` instructions.

- A non-transient data stream having unlimited length and which will access consecutive units in storage can be completely specified, including providing the hint that the program will probably soon access it, using one `dcbt` instruction. The corresponding specification for a data stream having other attributes requires two or three `dcbt/dcbtst` instructions to describe the stream and one additional `dcbt` instruction to start the stream. However, one `dcbt` instruction with TH=0b01010 and GO=1 can apply to a set of the data streams described in the preceding sentence, so the corresponding specification for n such data streams requires 2×n to 3×n `dcbt/dcbtst` instructions plus one `dcbt` instruction. (There is no need to execute a `dcbt/dcbtst` instruction with TH=0b01010 and S=0b10 for a given stream ID before using the stream ID for a new data stream; the implicit portion of the hint provided by `dcbt/dcbtst` instructions that describe data streams suffices.)

- If it is desired that the hint provided by a given `dcbt/dcbtst` instruction be provided in program order with respect to the hint provided by another `dcbt/dcbtst` instruction, the two instructions must be separated by an `eieio<S>` or `mbar<E>` instruction. For example, if a `dcbt` instruction with TH=0b01010 and GO=1 is intended to indicate that the program will probably soon access nascent data streams described (completely) by preceding `dcbt/dcbtst` instructions, and is intended not to indicate that the program will probably soon access nascent data streams described (completely) by following `dcbt/dcbtst` instructions, an `eieio<S>` or `mbar<E>` instruction must separate the `dcbt` instruction with GO=1 from the preceding `dcbt/dcbtst` instructions, and another `eieio<S>` or `mbar<E>` instruction must separate that `dcbt` instruction from the following `dcbt/dcbtst` instructions.

- In practice, the second `eieio<S>` or `mbar<E>` described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the `dcbt/dcbtst` instructions and an inner loop that contains the `Load` or `Store` instructions that access the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)

- To mitigate the effects of interrupts on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a virtual page.

- If it is desired to specify data streams without regard to the number of stream IDs provided by the implementation, stream IDs should be assigned to data streams in order of decreasing stream importance (stream ID 0 to the most important stream, stream ID 1 to the next most important stream, etc.). This order ensures that the hints for the most important data streams will be provided.

### TH=0b10000

If TH=0b10000, the `dcbt` instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the program's need for the block will be transient (i.e. the time interval during which the program accesses the block is likely to be short).

### TH=0b10001

If TH=0b10001, the `dcbt` instruction provides a hint that the program will probably not access the block containing the byte addressed by EA for a relatively long period of time.
Data Cache Block Allocate X-form

dcba RA,RB
[Category: Embedded]

Data Cache Block Touch X-form

dcbt RA,RB,TH [Category: Server]
dcbt TH,RA,RB [Category: Embedded]

Let the effective address (EA) be the sum (RA0)+(RB).

This instruction provides a hint that the program will probably soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. The contents of the block are undefined when the instruction completes. The hint is ignored if the block is Caching Inhibited.

This instruction is treated as a Store (see Section 4.3) except that the instruction is treated as a no-op if execution of the instruction would cause the system data storage error handler to be invoked.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics are provided for the Data Cache Block Touch instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

Extended: Equivalent to:
dcbtct RA,RB,TH dcbt for TH values of 0b00000 - 0b00111; other TH values are invalid.
dcbtds RA,RB,TH dcbt for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid.
dcbtt RA,RB dcbt for TH value of 0b10000
dcbna RA,RB dcbt for TH value of 0b10001
Data Cache Block Touch for Store X-form

dcbtst RA, RB, TH

dcbtst TH, RA, RB

Let the effective address (EA) be the sum (RA|0)+(RB).

The `dcbtst` instruction provides a hint that describes a block or data stream to which the program may perform a Store access, or indicates the expected use thereof. A hint that the program will soon store to a given storage location is ignored if the location is Caching Inhibited or Guarded.<S>

The only operation that is “caused by” the `dcbtst` instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be “caused by” or “associated with” the `dcbtst` instruction (e.g., `dcbtst` is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The `dcbtst` instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH≠0b01010 and TH≠0b01011, this instruction is treated as a Store (see Section 4.3), except that the system data storage error handler is not invoked, reference recording<S> need not be done, and change recording<S> is not done.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics are provided for the `Data Cache Block Touch for Store` instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

```
Extended:          Equivalent to:
dcbtstct RA, RB, TH  dcbtst for TH values of 0b00000
                   or 0b00000 - 0b00111;
                   other TH values are invalid.

dcbtstds RA, RB, TH   dcbtst for TH values of 0b00000
                   or 0b01000 - 0b01111;
                   other TH values are invalid.

dcbtstt RA, RB        dcbtst for TH value of 0b10000.
```

Programming Note

See the Programming Notes at the beginning of this section.
Data Cache Block set to Zero \quad X-form

dcbz \quad RA,RB

\begin{verbatim}
 0 31 ///  RA  RB   1014
 6

\end{verbatim}

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)
EA \( \leftarrow b + (RB) \)
n \( \leftarrow \) block size (bytes)
m \( \leftarrow \log_2(n) \)
ea \( \leftarrow EA_{0:63-m} \mid 0 \)
MEM(ea, n) \( \leftarrow \) n0x00

Let the effective address (EA) be the sum (RA0)+(RB).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a Store (see Section 4.3).

Special Registers Altered:

None

Programming Note

dcbz does not cause the block to exist in the data cache if the block is in storage that is Caching Inhibited.

For storage that is neither Write Through Required nor Caching Inhibited, dcbz provides an efficient means of setting blocks of storage to zero. It can be used to initialize large areas of such storage, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of Store instructions.

For storage that is either Write Through Required or Caching Inhibited, dcbz is likely to take significantly longer to execute than an equivalent sequence of Store instructions. For example, on some implementations dcbz for such storage may cause the system alignment error handler to be invoked; on such implementations the system alignment error handler sets the specified block to zero using Store instructions.

See Section 5.9.1 of Book III-S and Section 6.11.1 of Book III-E for additional information about dcbz.
Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a Load (see Section 4.3), except that reference and change recording need not be done, and it is treated as a Write with respect to debug events.

Special Registers Altered:
None

Let the effective address (EA) be the sum (RA|0)+(RB).

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 (“dcbf local”) [Category: Server, Embedded, Phased-In]

The L=1 form of the dcbf instruction permits a program to limit the scope of the “flush” operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 (“dcbf local primary”) [Category: Server, Embedded, Phased-In]

The L=3 form of the dcbf instruction permits a program to limit the scope of the “flush” operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

For the L operand, the value 2 is reserved. The results of executing a dcbf instruction with L=2 are boundedly undefined.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.
This instruction is treated as a Load (see Section 4.3), except that reference and change recording need not be done, and it is treated as a Write with respect to debug events.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics are provided for the Data Cache Block Flush instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A, "Assembler Extended Mnemonics" on page 827. The extended mnemonics are shown below.

Extended:                      Equivalent to:
dcbf RA, RB                   dcbf RA, RB, 0
      dcbfl RA, RB              dcbf RA, RB, 1
      dcbflp RA, RB           dcbf RA, RB, 3

Except in the dcbf instruction description in this section, references to "dcbf" in Books I-III imply L=0 unless otherwise stated or obvious from context; "dcbfl" is used for L=1 and "dcbflp" is used for L=3.

Programming Note

dcbf serves as both a basic and an extended mnemonic. The Assembler will recognize a dcbf mnemonic with three operands as the basic form, and a dcbf mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

Dcbf with L=1 can be used to provide a hint that a block in this processor's data cache will not be reused soon.

Dcbf with L=3 can be used to flush a block from the processor's primary data cache but reduce the latency of a subsequent access. For example, the block may be evicted from the primary data cache but a copy retained in a lower level of the cache hierarchy.

Programs which manage coherence in software must use dcbf with L=0.

4.3.2.1 Obsolete Data Cache Instructions [Category: Vector]

The Data Stream Touch (dstf), Data Stream Touch for Store (dstst), and Data Stream Stop (dss) instructions (primary opcode 31, extended opcodes 342, 374, and 822 respectively), which were proposed for addition to the Power ISA and were implemented by some processors, must be treated as no-ops (rather than as illegal instructions).

The treatment of these instructions is independent of whether other Vector instructions are available (i.e., is independent of the contents of MSRVEC<S> (see Book III-S) or MSRSPV (see Book III-E).

Programming Note

These instructions merely provided hints, and thus were permitted to be treated as no-ops even on processors that implemented them.

The treatment of these instructions is independent of whether other Vector instructions are available because, on processors that implemented the instructions, the instructions were available even when other Vector instructions were not.

The extended mnemonics for these instructions were dstt, dststt, and dssall.

4.3.3 “or” Instruction

“or” Cache Control Hint

or 26,26,26

This form of or provides a hint that stores caused by preceding Store and dcbz instructions should be performed with respect to other processors and mechanisms as soon as is feasible.
Extended Mnemonics:

Additional extended mnemonic for the or hint:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>miso</td>
<td>or 26,26,26</td>
</tr>
</tbody>
</table>

**Programming Note**

This form of the or instruction can be used to reduce latency in producer-consumer applications by requesting that modified data be made visible to other processors quickly.

**Producer:**

```assembly
addi r1, r1, 0x1234
sth r1, 0x1000(r0) # store data value 0x1234
addi r2, r2, 0x0001
stb r2, 0x1002(r0) # store nonzero flag byte
or r26, r26, r26  # miso
p_loop:
    lbz r2, 0x1002(r0)
    andi. r2, r2, 0x00FF
    bne p_loop        # wait for clear of flag byte
```

**Consumer:**

```assembly
c_loop:
    lbz r2, 0x1002(r0)
    andi. r2, r2, 0x00FF
    beq c_loop        # check for valid flag
    lbz r1, 0x1000(r0) # load data value
addi r2, r2, 0x0000
stb r2, 0x1002(r0) # clear valid flag
or r26, r26, r26  # miso
```

**Engineering Note**

Implementation of miso should attempt to order the hint relative to stores caused by preceding accesses. Implementations may elect to treat miso as a no-op.

Ways to implement miso include the following.

- In a store queue that has potentially different priorities for forwarding stores, the miso instruction could increase the priority of stores caused by preceding instructions.
- In a store queue that holds stores to allow for gathering, the miso instruction could release the hold and allow the stores to proceed to be performed with respect to other processors.

**Programming Note**

Warning: Other forms of or Rx,Rx,Rx that are not described in this section and in may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (ori 0,0,0) should be used.
4.4 Synchronization Instructions

The synchronization instructions are used to ensure that certain instructions have completed before other instructions are initiated, or to control storage access ordering, or to support debug operations.

4.4.1 Instruction Synchronize Instruction

**Instruction Synchronize**

*XL-form*

isync

|   | 19 | 6 | 11 | 16 | 21 | 150 | 31 |

Executing an `isync` instruction ensures that all instructions preceding the `isync` instruction have completed before the `isync` instruction completes, and that no subsequent instructions are initiated until after the `isync` instruction completes. It also ensures that all instruction cache block invalidations caused by `icbi` instructions preceding the `isync` instruction have been performed with respect to the processor executing the `isync` instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the `isync` instruction may complete before storage accesses associated with instructions preceding the `isync` instruction have been performed.

This instruction is context synchronizing (see Book III).

**Special Registers Altered:**

None

4.4.2 Load and Reserve and Store Conditional Instructions

The *Load And Reserve* and *Store Conditional* instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location. See Section 1.7.3, “Atomic Update” for additional information about these instructions.

The *Load And Reserve* and *Store Conditional* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1, “Fixed-Point Storage Access Instructions”, in Book I.

The storage location specified by the *Load And Reserve* and *Store Conditional* instructions must be in storage that is Memory Coherence Required if the location may be modified by another processor or mechanism. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler is invoked for the Server environment and may be invoked for the Embedded environment.

The *Load and Reserve* instructions include an Exclusive Access hint (EH), which can be used to indicate that the instruction sequence being executed is implementing one of two types of algorithms:

**Atomic Update (EH=0)**

This hint indicates that the program is using a fetch and operate (e.g., fetch and add) or some similar algorithm and that all programs accessing the shared variable are likely to use a similar operation to access the shared variable for some time.

**Exclusive Access (EH=1)**

This hint indicates that the program is attempting to acquire a lock and if it succeeds, will perform another store to the lock variable (releasing the lock) before another program attempts to modify the lock variable.

**Programming Note**

The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the reservation granule will cause the reservation created by the *Load And Reserve* instruction to be lost.
Programming Note

Because the Load And Reserve and Store Conditional instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, locking, etc.; see Appendix B) that are needed by application programs. Application programs should use these library programs, rather than use the Load And Reserve and Store Conditional instructions directly.

Programming Note

EH = 1 should be used when the program is obtaining a lock variable which it will subsequently release before another program attempts to perform a store to it. When contention for a lock is significant, using this hint may reduce the number of times a cache block is transferred between processor caches.

EH = 0 should be used when all accesses to a mutex variable are performed using an instruction sequence with Load and Reserve followed by Store Conditional (e.g., emulating atomic update primitives such as "Fetch and Add;" see Appendix B). The processor may use this hint to optimize the cache to cache transfer of the block containing the mutex variable, thus reducing the latency of performing an operation such as ‘Fetch and Add’.

Programming Note

Either value of the EH field is appropriate for a Load and Reserve instruction that is intended to establish a reservation for a subsequent waitrsv and not a subsequent Store Conditional instruction.

Programming Note

Warning: On some processors that comply with versions of the architecture that precede Version 2.00, executing a Load And Reserve instruction in which EH = 1 will cause the illegal instruction error handler to be invoked.

Load Byte And Reserve Indexed X-form

[Category: Phased-In]

lbarx RT,RA,RB,EH

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 6  | 11 | 16 | 21 | 31 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)

RESERVE ← 1
RESERVE_LENGTH ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← $560 || MEM(EA, 1)

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into RT₅₆:₆₃. RT₀:₅₅ are set to 0.

This instruction creates a reservation for use by a stbcx instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 1 byte is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the byte in storage addressed by EA regardless of the result of the corresponding stbcx instruction.
1 Other programs will not attempt to modify the byte in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

Special Registers Altered:
None

Programming Note

lbarx serves as both a basic and an extended mnemonic. The Assembler will recognize a lbarx mnemonic with four operands as the basic form, and a lbarx mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.
Load Halfword And Reserve Indexed X-form
[Category: Phased-In]

\[ \text{lharx } RT, RA, RB, EH \]

| if RA = 0 then b ← 0          
else                    b ← (RA)       
| EA ← b +(RB)             
| RESERVE ← 1             
| RESERVE_LENGTH ← 2      
| RESERVE_ADDR ← real_addr(EA)   
| RT ← 480 || MEM(EA, 2) |

Let the effective address (EA) be the sum (RA|0)+(RB).
The halfword in storage addressed by EA is loaded into RT48:63. RT0:47 are set to 0.

This instruction creates a reservation for use by a \text{sthcx}. instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 2 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the halfword in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the halfword in storage addressed by EA regardless of the result of the corresponding \text{sthcx}. instruction.
1 Other programs will not attempt to modify the halfword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:
None

Programming Note
\text{lharx} serves as both a basic and an extended mnemonic. The Assembler will recognize a \text{lharx} mnemonic with four operands as the basic form, and a \text{lharx} mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Word And Reserve Indexed X-form

\[ \text{lwarx } RT, RA, RB, EH \]

| if RA = 0 then b ← 0          
else                    b ← (RA)       
| EA ← b +(RB)             
| RESERVE ← 1             
| RESERVE_LENGTH ← 4      
| RESERVE_ADDR ← real_addr(EA)   
| RT ← 320 || MEM(EA, 4) |

Let the effective address (EA) be the sum (RA|0)+(RB).
The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

This instruction creates a reservation for use by a \text{stwcx}. instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 4 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the word in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the word in storage addressed by EA regardless of the result of the corresponding \text{stwcx}. instruction.
1 Other programs will not attempt to modify the word in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:
None

Programming Note
\text{lwarx} serves as both a basic and an extended mnemonic. The Assembler will recognize a \text{lwarx} mnemonic with four operands as the basic form, and a \text{lwarx} mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.
Store Byte Conditional Indexed X-form
[Category: Phased-In]

\textbf{stbcx}. RS,RA,RB

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>694</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>RA</td>
<td>b + (RB)</td>
<td>(RS)56:63</td>
<td>undefined_case</td>
<td>1</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
if RESERVE then
    if RESERVE_LENGTH = 1 then
        if RESERVE_ADDR = real_addr(EA) then
            MEM(EA, 1) ← (RS)56:63
            store_performed ← 1
        else
            undefined_case ← 1
    else
        if SCPM category supported then
            z ← smallest real page size supported by implementation
            if RESERVE_ADDR + z = real_addr(EA) + z then
                undefined_case ← 1
            else
                undefined_case ← 0
                store_performed ← 0
            else
                undefined_case ← 1
        else
            undefined_case ← 1
        store_performed ← 0
    if undefined_case then
        u1 ← undefined 1-bit value
        if u1 then
            MEM(EA, 1) ← (RS)56:63
            u2 ← undefined 1-bit value
            CR0 ← 0b00 || u2 || XERSO
        else
            CR0 ← 0b00 || store_performed || XERSO
        RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 1 byte, and the real storage location specified by the \textbf{stbcx}. is the same as the real storage location specified by the \textbf{lbarx} instruction that established the reservation, (RS)56:63 are stored into the byte in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 1 byte, it is undefined whether (RS)56:63 are stored into the byte in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

\textbf{CR0}_{LT \ GT \ EQ \ SO} = 0b00 \| n \| XERSO

The reservation is cleared.

\textbf{Special Registers Altered: CR0}
Store Halfword Conditional Indexed X-form
[Category: Phased-In]

sthcx. RS,RA,RB

if RA = 0 then b ← 0
else b ← (RA)

EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 2 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 2) ← (RS)48:63
      store_performed ← 1
    else undefined_case ← 1
  else
    undefined_case ← 0
    store_performed ← 0
else
  undefined_case ← 1
else undefined_case ← 1

if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    if MEM(EA, 2) ← (RS)48:63
    u2 ← undefined 1-bit value
    CR0 ← 0b00 || u2 || XER_S0
  else
    CR0 ← 0b00 || store_performed || XER_S0
  RESERVE ← 0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 2 bytes, and the real storage location specified by the sthcx. is the same as the real storage location specified by the lharx instruction that established the reservation, it is undefined whether (RS)48:63 are stored into the halfword in storage addressed by EA. Otherwise, no store is performed.

If the Store Conditional Page Mobility category is not supported, it is undefined whether (RS)48:63 are stored into the halfword in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 2 bytes, it is undefined whether (RS)48:63 are stored into the halfword in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

CR0_LT GT EQ SO = 0b00 || n || XER_S0

The reservation is cleared.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0
Store Word Conditional Indexed X-form

stwcx. RS,RA,RB

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>150</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
if RESERVE then
    if RESERVE_LENGTH = 4 then
        if RESERVE_ADDR = real_addr(EA) then
            (RS)32:63 are stored into the word in storage addressed by EA. Otherwise, no store is performed.
            - If the Store Conditional Page Mobility category is not supported, it is undefined whether (RS)32:63 are stored into the word in storage addressed by EA.
        If a reservation exists and the length associated with the reservation is not 4 bytes, it is undefined whether (RS)32:63 are stored into the word in storage addressed by EA.
        If a reservation does not exist, no store is performed.
        CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).
        CR0LT GT EQ SO = 0b00 || n || XERSO
        The reservation is cleared.
        EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.
        Special Registers Altered:
        CR0

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the stwcx. is the same as the real storage location specified by the lwarx instruction that established the reservation, (RS)32:63 are stored into the word in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the stwcx. is not the same as the real storage location specified by the lwarx instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size is the smallest real page size supported by the implementation. If the real storage location specified by the stwcx. is in the same z as the real storage location specified by the lwarx instruction that established the reservation, it is undefined whether

\[ (RS)32:63 \text{ are stored into the word in storage addressed by EA}. \]
### 4.4.2.1 64-Bit Load and Reserve and Store Conditional Instructions [Category: 64-Bit]

#### Load Doubleword And Reserve Indexed X-form

**ldarx RT,RA,RB,EH**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RT</td>
<td>RA</td>
<td>RB</td>
<td>84</td>
<td>EH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- if RA = 0 then b ← 0
- else b ← (RA)
- EA ← b + (RB)
- RESERVE ← 1
- RESERVE_LENGTH ← 8
- RESERVE_ADDR ← real_addr(EA)
- RT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

This instruction creates a reservation for use by a **stdcx**. instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 8 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding **stdcx** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**

None

#### Programming Note

**ldarx** serves as both a basic and an extended mnemonic. The Assembler will recognize a **ldarx** mnemonic with four operands as the basic form, and a **ldarx** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

---

#### Store Doubleword Conditional Indexed X-form

**stdcx RS,RA,RB**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>RA</td>
<td>RB</td>
<td>214</td>
<td></td>
</tr>
</tbody>
</table>

- if RA = 0 then b ← 0
- else b ← (RA)
- EA ← b + (RB)
- if RESERVE then
  - if RESERVE_LENGTH = 8 then
    - if RESERVE_ADDR = real_addr(EA) then
      - MEM(EA, 8) ← (RS)
    - undefined_case ← 0
    - store_performed ← 1
  - else
    - undefined_case ← 0
    - store_performed ← 0
  - else
    - undefined_case ← 1
  - else
    - undefined_case ← 1
  - else
    - undefined_case ← 0
    - store_performed ← 0
  - else
    - undefined_case ← 0
    - store_performed ← 0

Let the effective address (EA) be the sum (RA|0)+(RB). If a reservation exists, the length associated with the reservation is 8 bytes, and the real storage location specified by the **stdcx**. is the same as the real storage location specified by the **ldarx** instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 8 bytes, and the real storage location specified by the **stdcx**. is not the same as the real storage location specified by the **ldarx** instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size...
is the smallest real page size supported by the implementation. If the real storage location specified by the \texttt{stdcx} is in the same \texttt{z} as the real storage location specified by the \texttt{ldarx} instruction that established the reservation, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA. Otherwise, no store is performed.

- If the Store Conditional Page Mobility category is not supported, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 8 bytes, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. \(n\) is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of \(n\) is undefined (and need not reflect whether the store was performed).

\[
\text{CR}0_{\text{LT GT EQ SO}} = 0b00 \| n \| \text{XER}{\text{SO}}
\]

The reservation is cleared.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

**Special Registers Altered:**
4.4.2.2 CR0 128-bit Load and Reserve Store Conditional Instructions [Category: Load/Store Quadword]

For **lqarx**, the quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In big-endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

In the preferred form of the **Load Quadword** instruction RA ≠ RTp+1 and RB ≠ RTp+1.

For **stqcx**, the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In big-endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

### Load Quadword And Reserve Indexed X-form

**lqarx** RTp,RA,RB,EH

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>276</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>RTp</td>
<td>EB</td>
<td>RB</td>
<td>RA</td>
<td>RB</td>
<td>RA</td>
<td>RA</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0 else b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 16
RESERVE_ADDR ← real_addr(EA)
RTp ← MEM(EA, 16)

Let the effective address (EA) be the sum (RA|0)+(RB).
The quadword in storage addressed by EA is loaded into RTp.

This instruction creates a reservation for use by a **stqcx** instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 16 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding **stqcx** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 16. If it is not, the system alignment error handler is invoked.

If RTp is odd, RTp=RA, or RTp=RB the instruction form is invalid. If RTp=RA or RTp=RB, an attempt to execute this instruction will invoke the system illegal instruction error handler. (The RTp=RA case includes the case of RTp=RA=0.)

**Special Registers Altered:**
None

--- Programming Note ---

**lqarx** serves as both a basic and an extended mnemonic. The Assembler will recognize a **lqarx** mnemonic with four operands as the basic form, and a **lqarx** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.
Store Quadword Conditional Indexed X-form

\[ \text{stqcx.} \quad \text{RSp,RA,RB} \]

<table>
<thead>
<tr>
<th>31</th>
<th>RSp</th>
<th>RA</th>
<th>RB</th>
<th>182</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

EA \( \leftarrow b + (RB) \)

if RESERVE then
  if RESERVE_LENGTH = 16 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 16) \( \leftarrow (RSp) \)
      store_performed \( \leftarrow 1 \)
    else
      undefined_case \( \leftarrow 0 \)
      store_performed \( \leftarrow 0 \)
    else
      undefined_case \( \leftarrow 1 \)
  else
    undefined_case \( \leftarrow 0 \)
    store_performed \( \leftarrow 0 \)
else
  undefined_case \( \leftarrow 1 \)
else
  undefined_case \( \leftarrow 0 \)
store_performed \( \leftarrow 0 \)

if undefined_case then
  u1 \( \leftarrow \) undefined 1-bit value
if u1 then
  MEM(EA, 16) \( \leftarrow (RSp) \)
else
  undefined_case \( \leftarrow 0 \)
  store_performed \( \leftarrow 0 \)

u2 \( \leftarrow \) undefined 1-bit value
CR0 \( \leftarrow 0b00 \mid \mid u2 \mid \mid XER_{SO} \)

else
  CR0 \( \leftarrow 0b00 \mid \mid \text{store_performed} \mid \mid XER_{SO} \)
RESERVE \( \leftarrow 0 \)

Let the effective address (EA) be the sum (RAI0)+(RB).

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the \textit{stqcx.} is the same as the real storage location specified by the \textit{lqarx} instruction that established the reservation, (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the \textit{stqcx.} is not the same as the real storage location specified by the \textit{lqarx} instruction that established the reservation, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA. Otherwise, no store is performed.

- If the Store Conditional Page Mobility category is not supported, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 16 bytes, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. \( n \) is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of \( n \) is undefined (and need not reflect whether the store was performed).

\[ \text{CR0}_{LT \ GT \ EQ \ SO} = 0b00 \ll n \ll XER_{SO} \]

The reservation is cleared.

EA must be a multiple of 16. If it is not, the system alignment error handler is invoked.

If RSp is odd, the instruction form is invalid.

Special Registers Altered:

CR0
4.4.3 Memory Barrier Instructions

The Memory Barrier instructions can be used to control the order in which storage accesses are performed. See Section 1.8, “Transactions [Category: Transactional Memory]” for a description of how the Memory Barrier instructions interact with transactions. Additional information about these instructions and about related aspects of storage management can be found in Book III.

Extended mnemonics for Synchronize

Extended mnemonics are provided for the Synchronize instruction so that it can be supported by assemblers that recognize only the msync<E> mnemonic and so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A, “Assembler Extended Mnemonics” on page 827. The only reason for the msync<E> mnemonic is compatibility with Book E assembler code.

Synchronize X-form

<table>
<thead>
<tr>
<th>sync</th>
<th>L, E [Category: Elemental Memory Barriers]</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>L / E / 598 /</td>
</tr>
<tr>
<td>0</td>
<td>6 9 11 12 16 21 31</td>
</tr>
</tbody>
</table>

if E ≠ 0b0000 then
  if E = 0b1xxxx then enforce_barrier(mbll)
  if E = 0bx1xx then enforce_barrier(mbls)
  if E = 0bxx1x then enforce_barrier(mbsl)
  if E = 0bxxx1 then enforce_barrier(mbss)
else
  switch(L)
  case(0): hwsync
  case(1): lwsync
  case(2): ptesync<S> or hwsync<E>

The sync instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses that is ordered by the memory barrier depends on the value of the E and L fields. If the implementation supports the Elemental Memory Barriers category and the E field is nonzero, the L field is ignored as long as the E and L values do not create an invalid form (see Figure 4).

E=0 [Category: Elemental Barriers]

The memory barrier provides an ordering function for one or more distinct pairings of accesses to storage that is memory Coherence Required and is neither Write Through Required nor Caching Inhibited. Each bit in the E field that has a value of 1 selects pairings, as described below.

- E0=1 (mbll)
  The “load load” memory barrier is provided. Applicable pairs are all pairs a,b of such accesses in which a is an access caused by a Load instruction and b is an access caused by a Store or dcbz instruction.

- E1=1 (mbls)
  The “load store” memory barrier is provided. Applicable pairs are all pairs a,b of such accesses in which a is an access caused by a Load instruction and b is an access caused by a Store or dcbz instruction.

- E2=1 (mbsl)
  The “store load” memory barrier is provided. Applicable pairs are all pairs a,b of such accesses in which a is an access caused by a Store or dcbz instruction and b is an access caused by a Load instruction.

- E3=1 (mbss)
  The “store store” memory barrier is provided. Applicable pairs are all pairs a,b of such accesses in which both a and b are accesses caused by a Store or dcbz instruction.

L=0 (“heavyweight sync”)

The memory barrier provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the sync instruction. The applicable pairs are all pairs a,b of storage accesses in which b is a data access, except that if a is the storage access caused by an icbi instruction then b may be performed with respect to the processor executing the sync instruction before a is performed with respect to that processor.

L=1 (“lightweight sync”)

The memory barrier provides an ordering function for the storage accesses caused by Load, Store, and dcbz instructions that are executed by the processor executing the sync instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a,b of storage accesses except those in which a is an access caused by a Store or dcbz instruction and b is an access caused by a Load instruction.

L=2<S> (“ptesync”)

The set of storage accesses that is ordered by the memory barrier is described in Section 5.9.2 of
Book III-S, as are additional properties of the `sync` instruction with L=2.

The ordering done by the memory barrier is cumulative (regardless of the E and L values).

If L=0 (or L=2), the `sync` instruction has the following additional properties.

- Executing the `sync` instruction ensures that all instructions preceding the `sync` instruction have completed before the `sync` instruction completes, and that no subsequent instructions are initiated until after the `sync` instruction completes.
- The `sync` instruction is execution synchronizing (see Book III). However, address translation and reference and change recording (see Book III) associated with subsequent instructions may be performed before the `sync` instruction completes.
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a store in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an `icbi` instruction executed by the processor executing the `sync` instruction (P1) may not have been performed with respect to P1 (see the description of the `icbi` instruction on page 762).

The cumulative properties of the barrier apply to the execution of the given instruction as they would to a load that returned a value that was the result of a store in set B.

The value L=3 is reserved.

Figure 4 shows the valid encodings of the E and L fields, as well as the resulting memory barrier for systems that support the Elemental Memory Barriers category and those that do not.

---

**Programming Note**

In Figure 4, encodings in which E₂=1 also have L=0 (`hwsync`) to ensure backward compatibility.

---

**Assembler Note**

Combinations of E and L values other than those shown in Figure 4 should be flagged as errors.
The sync instruction may complete before storage accesses associated with instructions preceding the sync instruction have been performed.

See Section 6.11.3 of Book III-E for additional information related to sync with L=0 for the Embedded environment.

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics for Synchronize:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>sync 0</td>
</tr>
<tr>
<td>msync&lt;E&gt;</td>
<td>sync 0</td>
</tr>
<tr>
<td>lwsync</td>
<td>sync 1</td>
</tr>
<tr>
<td>ptesync&lt;S&gt;</td>
<td>ptesync&lt;S&gt; or hwsync&lt;E&gt;</td>
</tr>
<tr>
<td>ptesync&lt;S&gt;</td>
<td>ptesync&lt;S&gt; or hwsync&lt;E&gt;</td>
</tr>
</tbody>
</table>

Excerpt in the sync instruction description in this section, references to “sync” in Books I-III imply L=0 unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended. Throughout Books I-III, references to the L field imply E=0b0000 unless otherwise stated or obvious from context; the E field is mentioned explicitly when other E values are intended. Some programming examples and recommendations assume a common programming model that does not include the Elemental Memory Barriers category. Improved performance may be achieved through the use of elemental memory barriers in many cases.

Programming Note

Section 1.9 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

Programming Note

sync serves as both a basic and an extended mnemonic. The Assembler will recognize a sync mnemonic with one operand as the basic form, and a sync mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.
The sync instruction can be used to ensure that all stores into a data structure, caused by Store instructions executed in a “critical section” of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor; see Section B.2, “Lock Acquisition and Release, and Related Techniques” on page 831.

The memory barrier created by a sync instruction with L=1 does not order implicit storage accesses or instruction fetches. The memory barrier created by a sync instruction with L=0 (or L=2) orders implicit storage accesses and instruction fetches associated with instructions preceding the sync instruction but not those associated with instructions following the sync instruction.

In order to obtain the best performance across the widest range of implementations, the programmer should use the sync instruction with L=1, or the eleio<sub>S</sub> or mbar<sub>E</sub> instruction, if any of these is sufficient for his needs; otherwise he should use sync with L=0. sync with L=2<sub>S</sub> should not be used by application programs.

The functions provided by sync with L=1 are a strict subset of those provided by sync with L=0. (The functions provided by sync with L=2<sub>S</sub> are a strict superset of those provided by sync with L=0; see Book III.)
Enforce In-order Execution of I/O  X-form

eieio

[Category: Server]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>854</th>
<th>/</th>
</tr>
</thead>
</table>

The *eieio* instruction creates a memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses caused by *Load*, *Store*, *dcbez*, *eciwx*, and *ecowx* instructions executed by the processor executing the *eieio* instruction. These storage accesses are divided into the two sets listed below. The storage access caused by an *eciwx* instruction is ordered as a load, and the storage access caused by a *dcbez* or *ecowx* instruction is ordered as a store.

1. Loads and stores to storage that is both Caching Inhibited and Guarded, and stores to main storage caused by stores to storage that is Write Through Required.

   The applicable pairs are all pairs $a_i, b_j$ of such accesses.

2. Stores to storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited.

   The applicable pairs are all pairs $a_i, b_j$ of such accesses.

The operations caused by the stream variants of the *dcbt* and *dcbtst* instructions (i.e. the providing of hints) are ordered by *eieio* as a third set of operations, and the operations caused by *tibie* and *tibsinc* instructions (see Book III-S) are ordered by *eieio* as a fourth set of operations.

Each of the four sets of storage accesses or operations is ordered independently of the other three sets. The ordering done by *eieio*’s memory barrier for the second set is cumulative; the ordering done by *eieio*’s memory barrier for the other three sets is not cumulative.

The *eieio* instruction may complete before storage accesses associated with instructions preceding the *eieio* instruction have been performed. The *eieio* instruction may complete before operations caused by *dcbt* and *dcbtst* instructions preceding the sync instruction have been performed.

Special Registers Altered:

None

---

Memory Barrier  X-form

mbar  MO

[Category: Embedded]

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>854</th>
<th>/</th>
</tr>
</thead>
</table>

When MO=0, the *mbar* instruction creates a cumulative memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses executed by the processor executing the *mbar* instruction.

When MO≠0, an implementation may support the *mbar* instruction ordering a particular subset of storage accesses. An implementation may also support multiple, non-zero values of MO that each specify a different subset of storage accesses that are ordered by the *mbar* instruction. Which subsets of storage accesses that are ordered and which values of MO that specify these subsets is implementation-dependent.

The *mbar* instruction may complete before storage accesses associated with instructions preceding the *mbar* instruction have been performed. The *mbar* instruction may complete before operations caused by *dcbt* and *dcbtst* instructions preceding the sync instruction have been performed.

Special Registers Altered:

None

---

Programming Note

The *eieio*<S> and *mbar*<E> instructions are intended for use in doing memory-mapped I/O. Because loads, and separately stores, to storage that is both Caching Inhibited and Guarded are performed in program order (see Section 1.7.1, “Storage Access Ordering” on page 742), *eieio*<S> or *mbar*<E> is needed for such storage only when loads must be ordered with respect to stores.

For the *eieio*<S> instruction, accesses in set 1, $a_i$ and $b_j$ need not be the same kind of access or be to storage having the same storage control attributes. For example, $a_i$ can be a load to Caching Inhibited, Guarded storage, and $b_j$ a store to Write Through Required storage.

If stronger ordering is desired than that provided by *eieio*<S> or *mbar*<E>, the *sync* instruction must be used, with the appropriate value in the L field.

---

Power ISA™ - Book II
4.4.4 Wait Instruction

The wait instruction allows instruction fetching and execution to be suspended under certain conditions, depending on the value of the WC field. A **wait** instruction without the WC field is treated as a **wait** instruction with WC=0.

The defined values for WC are as follows.

- **0b00** Resume instruction fetching and execution when an interrupt occurs.
- **0b01** Resume instruction fetching and execution when an interrupt occurs or when a reservation made by the processor does not exist (see Section 1.7.3). It is implementation-dependent whether this WC value is supported or **wait** with this WC value is treated as a no-op.
- **0b10** Resume instruction fetching and execution when an interrupt occurs or when an implementation-specific condition exists. It is implementation-dependent whether this WC value is supported or is treated as reserved.
- **0b11** This WC value is treated as a no-op.

If WC=0, or if WC=1 and a reservation made by the processor exists when the wait instruction is executed, or if WC=2 and the associated implementation-specific condition does not exist when the **wait** instruction is executed, the following applies:

- Instruction fetching and execution is suspended.
- Once the **wait** instruction has completed, the NIA will point to the next sequential instruction.
- Instruction fetching and execution resumes when any of the following conditions are met.
  - An interrupt occurs.
  - WC=1 and a reservation made by the processor does not exist.
  - WC=2 and the associated implementation-specific condition exists.

---

Programming Note

The functions provided by **eieio**<S> for its second set are a strict subset of those provided by **sync** with L=1.

Since **eieio**<S> and **mbar**<E> share the same op-code, software designed for both Server and Embedded environments must assume that only the **eieio**<S> functionality applies since the functions provided by **eieio** are a subset of those provided by **mbar** with MO=0.
When the wait instruction is executed, if WC=1 and a reservation made by the processor does not exist, or if WC=2 and the associated implementation-specific condition exists, the instruction is treated as a no-op.

**Programming Note**
On implementations which do not support the wait instruction with WC=0b10, the behavior for non-support (treated as reserved) differs from the non-support of the other non-zero WC values (treated as no-ops). The possibility of boundedly undefined behavior such as causing the system illegal instruction error handler to be invoked is meant to discourage the use of WC=0b10 in programs that are intended to be portable.

Only programs that are implementation-aware should use WC=0b10.

**Engineering Note**
Causing the system illegal instruction error handler to be invoked if an attempt is made to execute wait with WC=0b10 on an implementation that does not support that form of wait facilitates the debugging of software.

**Special Registers Altered:**
None

**Extended Mnemonics:**
Examples of extended mnemonics for Wait:

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait</td>
<td>wait 0</td>
</tr>
<tr>
<td>waitrsv</td>
<td>wait 1</td>
</tr>
<tr>
<td>waitimpl</td>
<td>wait 2</td>
</tr>
</tbody>
</table>

**Programming Note**
The wait instruction with WC=0b00 can be used in verification test cases to signal the end of a test case. The encoding for the instruction is the same in both Big-Endian and Little-Endian modes.

**Programming Note**
The wait instruction may be useful as the primary instruction of an “idle process” or the completion of processing for a cooperative processor. However, overall system performance may be better served if the wait instruction is used by applications only for idle times that are expected to be short.

**Engineering Note**
In previous versions of the architecture the wait instruction was context synchronizing.
The *wait* instruction is not execution synchronizing and does not cause a memory barrier. *waitrsv* behavior relative to a preceding *Load and Reserve* instruction or *Store Conditional* instruction has a data dependency on the reservation. When execution proceeds past *waitrsv* as the result of another processor storing to the reservation granule, a subsequent load from the same storage location may return stale data. It is also possible that execution could proceed past the *waitrsv* for other reasons such as the occurrence of an interrupt. There are no architecturally defined means to determine what terminated the *wait*. Moreover, even if software were to attempt to determine what caused the *wait* to terminate, by the time the check occurred, both causes (interrupt and storage modification) might be true. Software must be designed to deal with the various causes of *wait* termination. In general, if the program that performed *wait* does not see the new value of the storage location for which the reservation was held, it should re-establish the reservation by repeating the *Load and Reserve* instruction, and then perform another *waitrsv*.

The following code waits for a device to update a memory location and assumes that r3 contains the address of the word to be updated. This assumes that software has already set this word to zero and is waiting for the device to update the word to a non-zero value.

```
loop:
    lwarx r4,0,r3 # load and reserve
    cmpwi r4,0    # exit if nonzero
    bne- exit
    waitrsv      # wait for reservation loss
    b loop
exit: ...
```

The *b* instruction results in re-execution of the *waitrsv* if instruction execution had resumed for some reason other than loss of the reservation made by the processor. This branch instruction is also necessary because the reservation might have been lost for reasons other than the device updating the memory location addressed by r3. Also, even if the device updated this memory location, the *lwarx* and *waitrsv* instructions may need to be re-executed until the *lwarx* returns the current data.

---

**Programming Note**

A *wait* instruction without the WC field is treated as a *wait* instruction with WC=0b00 because older processors that comply with Power ISA 2.06 do not support the WC field.
Chapter 5. Transactional Memory Facility [Category: Transactional Memory]

5.1 Transactional Memory Facility Overview

This chapter describes the registers and instructions that make up the transactional memory (TM) facility. Transactional memory is a shared-memory synchronization construct allowing an application to perform a sequence of storage accesses that appear to occur atomically with respect to other threads.

A set of instructions, special-purpose registers, and state bits in the MSR (see Book III) are used to control a transactional facility that is associated with each hardware thread. A \texttt{tbegin}. instruction is used to initiate transactional execution, and a \texttt{tend}. instruction is used to terminate transactional execution. Loads and stores that occur between the \texttt{tbegin}. and \texttt{tend}. instructions appear to occur atomically. An implementation may prematurely terminate transactional execution for a variety of reasons, rolling back all transactional storage updates that have been made by the thread since the \texttt{tbegin}. was executed, and rolling back the contents of a subset of the thread’s Book I registers to their contents before the \texttt{tbegin}. was executed. In the event of such premature termination, control is transferred to a software failure handler associated with the transaction, which may then retry the transaction or choose an alternate path depending on the cause of transaction failure. A transaction can be explicitly aborted via a set of \texttt{conditional abort} instructions and an \texttt{unconditional abort} instruction, \texttt{tabort}. A \texttt{tsr}. instruction is used to suspend or resume transactional execution, while allowing the transaction to remain active.

---

Programming Note

A \texttt{tbegin}. should always be followed immediately by a \texttt{beq} as the first instruction of the failure handler, that branches to the main body of the failure handler. The failure handler should always either retry the transaction or use non-transactional code to perform the same operation. (The number of retries should be limited to avoid the possibility of an infinite loop. The limit could be based on the perceived permanence / transience of the failure.)

A failure handler policy which includes trying a different transaction before returning to the one that failed may fail to make forward progress.

---

Programming Note

In code that may be executed transactionally, conditional branches should hint in favor of successful transactional execution where such a distinction exists. For example, the branch immediately following \texttt{tbegin}. should hint that the branch is very likely not to be taken. As another example, consider a method of coding a failure handler that executes the body of a transaction non-transactionally by branching past the TM control instructions (e.g. \texttt{tsuspend}). Branches that bypass the TM control instructions should also hint that the branch is very likely not to be taken. These predictions will improve the efficiency of transactional execution, and may also help prevent the addition of spurious accesses to the transactional footprint.
Transactions performed using this facility are “strongly atomic,” meaning that they appear atomic with respect to both transactional and non-transactional accesses performed by other threads. Transactions are isolated from reads and writes performed by other threads; i.e., transactional reads and writes will not appear to be interleaved with the reads and writes of other threads.

Nesting of transactions is supported using a form of nesting called “flattened nesting,” in which transactions that are initiated during transactional execution are subsumed by the pre-existing transaction. Consequently, the effects of a nested transaction do not become visible until the outer transaction commits, and if a nested transaction fails, the entire set of transactions (outer as well as nested) is rolled back, and control is transferred to the outer transaction’s failure handler. The memory barriers created by `tbegin`, and `tend`, and the integrated cumulative memory barrier that are described in Section 1.8, “Transactions [Category: Transactional Memory]” are only created for outer transactions and not for any transactions nested within them.

References to `Store` instructions, and stores, include `dcbz` and the storage accesses that it causes.

Rollback-Only Transactions

Rollback-Only Transactions (ROTs) differ from normal transactions in that they are speculative but not atomic. They may be nested with other ROTs or with normal transactions. When a normal transaction is initiated within a ROT, the behavior from the normal `tbegin`, until the end of the outer transaction is characteristic of a normal transaction. Although subject to failure from storage conflicts, the typical cause of ROT failure is via a `Tabort` variant that is executed after the program detects an error in its (software) speculation. Except where specifically differentiated or where differences follow from specific differentiation, the following description applies to ROTs as well as normal transactions.

5.1.1 Definitions

Commit: A transaction is said to commit when it successfully completes execution. When a transaction is committed, its transactional accesses become irrevocable, and are made visible to other threads. A transaction completes by either committing or failing.

Checkpointed registers: The set of registers that are saved to the “checkpoint area” when a transaction is initiated, and restored upon transaction failure, is a subset of the architected register state, consisting of the General Purpose Registers, Floating-Point Registers, Vector Registers, Vector-Scalar Registers, and the following Special Registers and fields: CR fields other than CR0, LR, CTR, FPSCR, AMR, PPR, VRSAVE, VSCR, DSCR, and TAR. The checkpointed registers include all problem-state writable registers with the exception of CR0, EBBHR, EBBRR, BESCR, the performance monitor registers, and the Transactional Memory registers. With the exception of updates of CR0, and the Transactional Memory registers, explicit updates of registers that are not included in the set of checkpointed registers are disallowed in Transactional state (i.e. will cause the transaction to fail), but are permitted in Suspended state. Suspended state modifications of these registers will not be rolled back in the event of transaction failure. (Modifications of Transactional Memory registers are only permitted in Non-transactional state. Attempts to modify Transactional Memory registers in other than Non-transactional state will cause a TM Bad Thing type Program interrupt.)

CR0, and the Transactional Memory registers (TFHAR, TEXASR, TFIAR) are not saved, or restored when the transaction fails, because they are modified as a side effect of transaction failure (so restoring them would lose information needed by the failure handler). The performance monitor registers, and the event-based branching registers (BESCR, EBBHR, EBBRR) are not saved or restored because saving and restoring them would add significant implementation complexity and is not needed by software. Also, these registers, except EBBHR, can be modified asynchronously by the processor, so restoring them when the transaction fails could cause loss of information.

Transactional accesses: Data accesses that are caused by an instruction that is executed when the thread is in the Transactional state (see Section 5.2) are said to be “transactional,” or to have been “performed transactionally.” The set of accesses caused by a committed normal transaction is performed as if it were a single atomic access. That is, it is always performed in its entirety with no visible fragmentation. The sets performed by normal transactions are thus serialized: each happens in its entirety in some order, even
when that order is not specified in the program or enforced between processors. Until a transaction commits, its set of transactional accesses is provisional, and will be discarded should the transaction fail. The set of transactional accesses is also referred to as the “transactional footprint.”

**Non-transactional accesses**: Storage accesses performed in the existing Power storage model are said to be “non-transactional.” In contrast to transactional storage accesses, there is no provision of atomicity across multiple non-transactional accesses. Non-transactional storage updates are not discarded in the event of a transaction failure.

**Outer transaction**: A transaction that is initiated from the Non-transactional state is said to be an outer transaction. A `tbegin` instruction that initiates an outer transaction is sometimes referred to as an “outer `tbegin`.” Similarly, a `tend` instruction with A=0 that ends an outer transaction is sometimes referred to as an “outer `tend`.”

**Nested Transaction**: A transaction that is initiated while already executing a transaction is said to be “nested” within the pre-existing transaction. The set of active nested transactions forms a stack growing from the outer transaction. A `tend` with A=0 will remove the most recently nested transaction from the stack.

**Failure**: A transaction failure is an exceptional condition causing the transactional footprint to be discarded, the checkpointed registers to be reverted to their pre-transactional values, and the failure handler to get control.

**Failure handler**: A failure handler is a software component responsible for handling transaction failure. On transaction failure, hardware redirects control to the failure handler associated with the outer transaction.

**Conflict**: A transactional memory access is said to conflict with another transactional or non-transactional access if both accesses reference the same cache block, and at least one of them is a store. If two transactions make conflicting accesses, at least one of them will fail. If a transaction fails as a result of a conflict with a store, the store may have been executed by another processor or may have been executed in Suspended state by the processor with the failing transaction. For a ROT, no conflict is caused if the ROT performs a load and another program performs a non-transactional store to the same block.

A transactional memory access is said to conflict with a `tlbie` if the storage location being accessed is in the page the translation for which is being invalidated by the tlbie. For a ROT, no conflict is caused if the access is a load.

A Suspended state cache control instruction is said to cause a conflict if it would cause the destruction of a transactional update or if it would make a transactional update visible to another thread.

---

### 5.2 Transactional Memory Facility States

The transactional memory facility supports several modes of operation, referred to in this document as the “transaction state.” These states control the behavior of storage accesses made during the transaction and the handling of transaction failure. Changes to transaction state affect all transactions currently using the transactional facility on the affected thread: the outer transaction as well as any nested transactions, should they exist.

**Non-transactional**: The default, initial state of execution; no transaction is executing. The transactional facility is available for the initiation of a new transaction.

**Transactional**: This state is initiated by the execution of a `tbegin` instruction in the Non-transactional state. Storage accesses (data accesses) caused by instructions executed in the Transactional state are performed transactionally. Other storage accesses associated with instructions executed in the Transactional state (instruction fetches, implicit accesses) are performed non-transactionally. In the event of transaction failure, failure is recorded as defined in Section 5.3.2, and control is transferred to the failure handler as described in Section 5.3.3.

**Suspended**: The Suspended execution state is explicitly entered with the execution of a `tsuspend` form of `tsr` instruction during a transaction, the execution of a `trechkpt` instruction from Non-transactional state, or as a side-effect of an interrupt while in the Transactional state. Storage accesses and accesses to SPRs that are not part of the checkpointed registers are performed non-transactionally; they will be performed independently of the outcome of the transaction. The initiation of a new transaction is prevented in this state. In the event of transaction failure, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred until transactional execution is resumed (see Section 5.3.3 for details).

Until failure occurs, `Load` instructions that access storage locations that were transactionally written by the same thread will return the transactionally written data. After failure is detected, but before failure handling is performed, such loads may return either the transactionally written data, or the current non-transactional contents of the accessed location. The `tcheck` instruction can be used to determine whether any previous such loads may have returned non-transactional contents.

Suspended state `Store` instructions that access cache blocks that have been accessed transactionally (due to load or store) by the same thread may cause the transaction to fail.
Table 1 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the \texttt{rfebb}, \texttt{rfid}, \texttt{hrfid}, and \texttt{mtmsrd} instructions, can also cause changes to the transaction state; see Book III.)

The intent of the Suspended execution state is to temporarily escape from transactional handling when transactional semantics are undesirable. Examples of such cases include storage updates that should be retained in the event of transactional failure, which is useful for debugging, interthread communication, the access of Caching-Inhibited storage, and the handling of interrupts. In the event of transaction failure during the Suspended execution state, failure handling is deferred until transactional execution is resumed, allowing the block of Suspended state code to complete its activities.

During Suspended state execution, accessing cache blocks that have been transactionally accessed by the same thread prior to entering Suspended state requires special care, because failure may occur due to uncontrollable events such as interactions with other threads or the operating system. Up until a transaction fails, loads from transactionally modified cache blocks will return the transactionally modified data. However once the transaction fails, the loads may return either the transactionally updated version of storage, or the most recent non-transactional version. Stores to transactionally modified blocks may or may not cause the thread's transaction to fail.

Table 1 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the \texttt{rfebb}, \texttt{rfid}, \texttt{hrfid}, and \texttt{mtmsrd} instructions, can also cause changes to the transaction state; see Book III.)

\textbf{Programming Note}

The intent of the Suspended execution state is to temporarily escape from transactional handling when transactional semantics are undesirable. Examples of such cases include storage updates that should be retained in the event of transactional failure, which is useful for debugging, interthread communication, the access of Caching-Inhibited storage, and the handling of interrupts. In the event of transaction failure during the Suspended execution state, failure handling is deferred until transactional execution is resumed, allowing the block of Suspended state code to complete its activities.

During Suspended state execution, accessing cache blocks that have been transactionally accessed by the same thread prior to entering Suspended state requires special care, because failure may occur due to uncontrollable events such as interactions with other threads or the operating system. Up until a transaction fails, loads from transactionally modified cache blocks will return the transactionally modified data. However once the transaction fails, the loads may return either the transactionally updated version of storage, or the most recent non-transactional version. Stores to transactionally modified blocks may or may not cause the thread's transaction to fail.

Table 1 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the \texttt{rfebb}, \texttt{rfid}, \texttt{hrfid}, and \texttt{mtmsrd} instructions, can also cause changes to the transaction state; see Book III.)

\textbf{Programming Note}

\texttt{tbegin}. in Suspended state merely updates CR0. When \texttt{tbegin}. is followed by \texttt{beq}. this will result in a transfer to the failure handler. Nothing more severe (e.g. an interrupt) is required. The failure handler for a transaction for which initiation may be attempted in Suspended state should test CR0 to determine whether \texttt{tbegin}. was executed in Suspended state. If so, it should attempt to emulate the transaction non-transactionally. (This case can arise, for example, if a transaction enters Suspended state and then calls a library routine that independently attempts to use transactions.)

Notice that, although a failure handler runs in Non-transactional state when reached because the transaction has failed, it runs in Suspended state for the case discussed in this Programming Note.)
Chapter 5. Transactional Memory Facility [Category: Transactional Memo-
Transactions will fail for the following self-induced causes:

- Termination caused by the execution of `tabort, tabortdc, tabortdc1, tabortwc, tabortwci` or `treclaim` instruction.
- Transaction level overflow, defined as an attempt to perform a storage access in Transactional state which exceeds the capacity for tracking transactional accesses.
- Execution of the following instructions while in the Transactional state: `doze, sleep, nap, rwhinkle, icbi, dcbf, dcbi, dcbst, [h]rfid, rfebb, mtmsr[d], mtse, mtsr, mtstn, msgsnd, msgrnd, msgclr, msgclrp, slbie, slbia, slbmt, slbfee`, and `tlbie[l]`. (These instructions are considered to be disallowed in Transactional state.) The disallowed instruction is not executed; failure handling occurs before it has been executed.

### Programming Note

Note that execution of a Power Saving instruction in Suspended state causes a TM Bad Thing type Program interrupt.

Execution, while in Transactional state, of `mtspr` specifying an SPR that is not part of the checkpointed registers and is not a Transactional Memory SPR. The `mtspr` is not executed; failure handling occurs before it has been executed. (Modification of XERFXCC and CPRX0 are allowed, but the changes will not be rolled back in the event of transaction failure.)

- Conflict caused by a Suspended state store to a block that was previously accessed transactionally. If the store would have been performed by a successful `Store Conditional` instruction, but the `Store Conditional` instruction does not succeed, it is implementation-dependent whether a conflict is detected.
- Conflict caused by a Suspended state `tlbie` that specifies a translation that was previously used transactionally. (This case will be recorded as a translation invalidation conflict because it may be hard to differentiate from a conflict caused by a `tlbie` performed by another thread and because it is highly likely to be a transient failure.)

For each of the following potential causes, the transaction will fail if the absence of failure would compromise transaction semantics; otherwise, whether the transaction fails is undefined.

- Execution of the following instructions while in the Transactional state: `ecwx, ecowx, lbzci, ldci, lhzci, lwzci, stbci, stdci, sthcix, stwci`. The disallowed instruction is not executed; failure handling occurs before it has been executed. (These instructions are considered to be disallowed in Transactional state if they cause transaction failure in Transactional state.) Execution of these instructions in the Suspended state is allowed and does not cause transaction failure.
- Execution of the following instruction in the Transactional state: `wait`. The disallowed instruction is not executed; failure handling occurs before it has been executed. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Execution of the following instruction in the Suspended state: `wait`. The disallowed instruction is treated as a no-op; failure recording occurs. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Access of a disallowed type while in the Transactional state: Caching Inhibited, Write Through Required, and Memory Coherence not Required for data access; Caching-Inhibited for instruction fetch. The disallowed access is not performed; failure handling occurs such that the instruction that would cause (or be associated with, for instruction fetch) the disallowed access type appears not to have been executed. Accesses of this type in the Suspended state are allowed and do not cause transaction failure.
- Instruction fetch from a block that was previously written transactionally (reported as a unique cause that includes both self-induced and externally-induced instances)

### Programming Note

Note that `dcbi` with L=3 should never compromise transactional semantics, but it is still permitted to cause transaction failure in Suspended state and it is disallowed in Transactional state.

The instruction (`dcbf, dcbi, or icbi`) is executed in Suspended state on the processor executing the transaction (self-induced conflict)
- the instruction is executed by another processor (externally-induced conflict)
- `dcbst` specifying a block that was previously written transactionally, in either of the following cases.

For each of the following potential causes, the transaction will fail if the absence of failure would compromise transaction semantics; otherwise, whether the transaction fails is undefined.
dcbst is executed by another processor (externally-induced conflict)
Cache eviction of a block that was previously accessed transactionally
Transactions may also fail due to implementation-specific characteristics of the transactional memory mechanism.

__Programming Note__
WARNING: Software should not depend for its correct execution on the behavior (whether or not the relevant transaction fails) of the cases described in the preceding set of bullets. The behavior is likely to vary from design to design. Such a dependence would impact the software's portability without any tangible advantage.

__Programming Note__
Because the atomic nature of a transaction implies an apparent delay of its component accesses until they can be performed in unison, the use of cache control instructions to manage cache residency and/or the performing of storage accesses may have unexpected consequences. Although they may not cause transaction failure directly, their use in a transaction is strongly discouraged.

If an instruction or event does not cause transaction failure, it behaves as defined in the architecture.

The set of failure causes and events are further classified as "precise" and "imprecise" failure causes. All externally induced events are imprecise, and all self-induced events are precise with the exception of the following cases:
- Self-induced conflicts caused by instruction fetch
- Self-induced conflicts caused by footprint overflow
- Self-induced conflicts in Suspended state (because failure handling is deferred in Suspended state).

When failure recording and handling occur (as defined in Section 5.3.2 and 5.3.3) for a precise failure, they will occur precisely according to the sequential execution model, adhering to the following rules:

1. Effects of the failure occur such that all instructions preceding the instruction causing the failure appear to have completed with respect to the executing thread.
2. The instruction causing the failure may appear not to have begun execution (except for causing the failure), or may have completed, depending on the failure cause.
3. Architecturally, no subsequent instruction has begun execution.

Failure handling for imprecise failure types is guaranteed to occur no later than failure handling. Any operation that can cause imprecise failure if performed in-order can also cause imprecise failure if performed out-of-order.

__Programming Note__
Because instruction fetch from a transactionally written block may result in failure, it is recommended that transactionally accessed data and transactionally accessed instructions not be co-located within a single block.

__Programming Note__
The architecture does not detect and cause transaction failure for translation invalidations to transactionally accessed pages or segments, when the translation invalidation is caused by instructions other than `tlbie` (i.e. `slbie`, `sliba`, `tlibel`, `tlbia`). Consequently, software is responsible for terminating transactions in circumstances where such local translation invalidations may affect a local transaction.

### 5.3.2 Recording of Transaction Failure

When transaction failure occurs, information about the cause and circumstances of failure are recorded in SPRs associated with the transactional facility. Failure recording is performed a single time per transaction that fails, controlled by the state of the TEXASR failure summary (FS) bit; when 0, FS indicates that failure recording has not already been performed, and is therefore permissible.

The following RTL function specifies the actions taken during the recording of transaction failure:

```
TMRecordFailure(FailureCause) #FailureCause is 32-bit cause code
if TEXASRFs = 0
  if failure IA known then
    TFIAR ← CIA
    TEXASRy ← 1
  else
    TFIAR ← approximate instruction address
    TEXASRy ← 0
  TEXASR0:31 ← FailureCause
  if MRSRy=0b01 then TEXASRsuspended ← 1
  if environment = Embedded then
    TEXASRPRIVILEGE ← ¬MRSRy || MSRPR
    TFIARPRIVILEGE ← ¬MRSRy || MSRPR
  else
    TEXASRPRIVILEGE ← MRSRy || MSRPR
    TFIARPRIVILEGE ← MRSRy || MSRPR
  TEXASRP ← 1
  TDOOMED ← 1
```

WARNING: Software should not depend for its correct execution on the behavior (whether or not the relevant transaction fails) of the cases described in the preceding set of bullets. The behavior is likely to vary from design to design. Such a dependence would impact the software’s portability without any tangible advantage.

Because instruction fetch from a transactionally written block may result in failure, it is recommended that transactionally accessed data and transactionally accessed instructions not be co-located within a single block.

The architecture does not detect and cause transaction failure for translation invalidations to transactionally accessed pages or segments, when the translation invalidation is caused by instructions other than `tlbie` (i.e. `slbie`, `sliba`, `tlibel`, `tlbia`). Consequently, software is responsible for terminating transactions in circumstances where such local translation invalidations may affect a local transaction.

### 5.3.2 Recording of Transaction Failure

When transaction failure occurs, information about the cause and circumstances of failure are recorded in SPRs associated with the transactional facility. Failure recording is performed a single time per transaction that fails, controlled by the state of the TEXASR failure summary (FS) bit; when 0, FS indicates that failure recording has not already been performed, and is therefore permissible.

The following RTL function specifies the actions taken during the recording of transaction failure:

```
TMRecordFailure(FailureCause) #FailureCause is 32-bit cause code
if TEXASRFs = 0
  if failure IA known then
    TFIAR ← CIA
    TEXASRy ← 1
  else
    TFIAR ← approximate instruction address
    TEXASRy ← 0
  TEXASR0:31 ← FailureCause
  if MRSRy=0b01 then TEXASRsuspended ← 1
  if environment = Embedded then
    TEXASRPRIVILEGE ← ¬MRSRy || MSRPR
    TFIARPRIVILEGE ← ¬MRSRy || MSRPR
  else
    TEXASRPRIVILEGE ← MRSRy || MSRPR
    TFIARPRIVILEGE ← MRSRy || MSRPR
  TEXASRP ← 1
  TDOOMED ← 1
```
When failure recording occurs, the TEXASR and TFIAR SPRs are set indicating the source of failure. When possible, TFIAR is set to the effective address of the instruction that caused the failure, and TEXASR₃₇ is set to 1 indicating that the contents of TFIAR are exact. When the instruction address is not known exactly, an approximate value is placed in TFIAR and TEXASR₃₇ is set to 0. TEXASR bits 0:31 are set indicating the cause of the failure, and the TEXASRₜₚₑₓₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑₚₑₑ₢
TFHAR stores the effective address of the software failure handler used in the event of transaction failure. TFIAR is used to inform software of the exact location of the transaction failure, when possible. TEXASR contains a transaction level indicating the nesting depth of an active transaction, as well as an indicator of the cause of transaction failure and some machine state when the transaction failed. These registers can be written only when in Non-transactional state.

5.4.1 Transaction Failure Handler Address Register (TFHAR)

The Transaction Failure Handler Address Register is a 64-bit SPR that records the effective address of a software failure handler used in the event of transaction failure. Bits 62:63 are reserved.

---

**Programming Note**

The Failure Persistent bit may be viewed as an eighth bit in the failure code in that both fields are supplied by the least significant byte of RA and software may use all eight to differentiate among the cases for which it performs an abort or reclaim. However, software is expected to organize its cases so that bit 7 predicts the persistence of the case.

---

**Warning:** Software must not depend on the value of the Failure Persistent bit for correct execution. The number of retries for a transient failure should be counted, and a limit set after which the program will perform the operation non-transactionally. In the analysis of failures, consideration should be given to the fact that speculative execution can cause unexpected behavior.

---

**Programming Note**

An instruction fetch to storage that is Caching Inhibited, while nominally disallowed, will be reported as Implementation-specific (bit 15). This choice was made because it seems like a relatively unlikely programming error, and there is a significant chance that data from an external conflict (store by another thread) could indirectly cause a wild branch to storage that is Caching Inhibited.

---

**Disallowed**
The instruction, SPR, or access type is not permitted. When set, TFIAR is exact. See Section 5.3.1, “Causes of Transaction Failure”.

---

**Programming Note**

The maximum transaction level was exceeded. When set, TFIAR is exact.

---

**Footprint Overflow**
The tracking limit for transactional storage accesses was exceeded. When set, TFIAR is an approximation.
Self-Induced Conflict
A self-induced conflict occurred in Suspended state, due to one of the following: a store to a block that was previously accessed transactionally; a dcbf, dcbi, or icbi specifying a block that was previously accessed transactionally; a dcbst specifying a block that was previously written transactionally; or a tblie that specifies a translation that was previously used transactionally. When set, TFIAR may be exact.

Non-Transactional Conflict
A conflict occurred with a non-transactional access by another processor. When set, TFIAR is an approximation.

Transaction Conflict
A conflict occurred with another transaction. When set, TFIAR may be exact.

Translation Invalidation Conflict
A conflict occurred with a TLB invalidation. When set, TFIAR is an approximation.

Implementation-specific
An implementation-specific condition caused the transaction to fail. Such conditions are transient and the value in the TFIAR may be exact.

Instruction Fetch Conflict
An instruction fetch (by this or another thread) was performed from a block that was previously written transactionally. Such conditions are transient and the value in the TFIAR may be exact.

Abort
Termination was caused by the execution of a tabort, tabortdc, tabortdci, tabortwc, tabortwci, or treclaim, instruction. When due to tabort, or treclaim, bits in TEXASR0:7 are user-supplied. When set, TFIAR is exact.

Suspended
When set to 1, the failure was recorded in Suspended state. When set to 0, the failure was recorded in Transactional state.

Reserved

Privilege
The thread was in this privilege state when the failure was recorded. For the Embedded environment, this was the value ¬MSRGS II MSRPR when the failure was recorded. For the Server environment, this was the value MSRIV II MSRPR when the failure was recorded.

Failure Summary (FS)
Set to 1 when a failure has been detected and failure recording has been performed.

TFIAR Exact
Set to 1 when the value in the TFIAR is exact. Otherwise the value in the TFIAR is approximate.

ROT
Set to 1 when a ROT is initiated. Set to zero when a non-ROT tbegin, is executed.

Transaction Level (TL)
Transaction level (nesting depth + 1) for the active transaction, if any; otherwise 0 if the most recently executed transaction completed successfully, or the transaction level at which the most recently executed transaction failed if the most recently executed transaction did not complete successfully.

Programming Note
A value of 1 corresponds to an outer transaction. A value greater than 1 corresponds to a nested transaction.

The transaction level in TEXASRtl contains an unsigned integer indicating whether the current transaction is an outer transaction, or is nested, and if nested, its depth. The maximum transaction level supported by a given implementation is of the form 2^t - 1. The value of t corresponding to the smallest maximum is 4; the value of t corresponding to the largest maximum is 12. This value is tied to the “Maximum transaction level” parameter useful for application programmers, as specified in Section 4.1. The high-order 12-t bits of TEXASRtl are treated as reserved.

Transaction failure information is contained in TEXASR0:37. The fields of TEXASR are initialized upon the successful initiation of a transaction from the Non-transactional state, by setting TEXASRtl to 1, indicating an outer transaction, and all other fields to 0.

When transaction failure is recorded, the failure summary bit TEXASRFS is set to 1, indicating that failure has been detected for the active transaction and that failure recording has been performed. TEXASR0:31 are set indicating the source of the failure. Exactly one of bits 8 through 31 will be set indicating the instruction or event that caused failure. In the event of failure due to
the execution of a `tabort`, `tabortdc`, `tabortdci`, `tabortwc`, `tabortwci`, or `treclaim` instruction, `TEXASR31` is set to 1, and, for `tabort` and `treclaim`, a software defined failure code is copied from a register operand to `TEXASR0:7`. `TEXASRSuspended` indicates whether the transaction was in the Suspended state at the time that failure occurred. The inverse of the value of MSRGS and the value of MSRPR for the Embedded environment or the values of MSRHV and MSRPR for the Server environment at the time that failure occurs are copied to `TEXASR34` and `TEXASR35`, respectively. In some circumstances, the failure causing instruction address in TFIAR may not be exact. In such circumstances, `TEXASR37` is set to 0 indicating that the contents of TFIAR are not exact; otherwise `TEXASR37` is set to 1.

5.4.3 Transaction Failure Instruction Address Register (TFIAR)

The Transaction Failure Instruction Address Register is a 64-bit SPR that is set to the exact effective address of the instruction causing the failure, when possible. Bits 62:63 contain the privilege state when the instruction causing the failure, when possible. For the Embedded environment, this was the value `¬MSRGS` || `MSRPR` when the failure was recorded. For the Server environment, this was the value `MSRHV` || `MSRPR` when the failure was recorded. For the Server environment, this was the value `MSRHV` || `MSRPR` when the failure was recorded.

<table>
<thead>
<tr>
<th>TFIAR</th>
<th>Privilege</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8. Transaction Failure Instruction Address Register (TFIAR)

In certain cases, the exact address may not be available, and therefore TFIAR will be an approximation. An approximate value will point to an instruction near the instruction that was executing at the time of the failure. TFIAR accuracy is recorded in an Exact bit residing in `TEXASR37`.

Programming Note

The transaction level contained in `TEXASR_TL` should be interpreted by software as follows:

When in the Transactional or Suspended state, this field contains an unsigned integer representing the transaction level of the active transaction, with 1 indicating an outer transaction, and a number greater than 1 indicating a nested transaction. The nesting depth of the active transaction is `TEXASR_TL` – 1.

When in the Non-transactional state, `TEXASR_TL` contains 0 if the last transaction committed successfully, otherwise it contains the transaction level at which the most recent transaction failed.

Programming Note

The Privilege bits in `TEXASR` represent the state of the machine at the point when failure occurs. This information may be used by problem-state software to determine whether an unexpected hypervisor or operating system interaction was responsible for transaction failure. This information may be useful to operating systems or hypervisors when restoring register state for failure handling after the transactional facility was reclaimed, to determine which of the operating system or the hypervisor has retained the pre-transactional version of the checkpointed registers.

5.4.3 Transaction Failure Instruction Address Register (TFIAR)

The Transaction Failure Instruction Address Register is a 64-bit SPR that is set to the exact effective address of the instruction causing the failure, when possible. Bits 62:63 contain the privilege state when the instruction causing the failure, when possible.
5.5 Transactional Facility Instructions

Similar to the Floating-Point Status and Control Register instructions, modifications of transaction state caused by the execution of Transactional Memory instructions or by failure handling synchronize the effects of exception-causing floating-point instructions executed by a given processor. Executing a Transactional Memory instruction, or invocation of the failure handler, ensures that all floating-point instructions previously initiated by the given processor have completed before the transaction state is modified, and that no subsequent floating-point instructions are initiated by the given processor until the transaction state has been modified. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the transaction state is modified.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the transaction state is modified.
- No subsequent floating-point instruction that alters the settings of any FPSCR bits is initiated until the transaction state has been modified.

(Floating-point Storage Access instructions are not affected.)

Transaction Begin X-form

tbegin. R

<table>
<thead>
<tr>
<th>31</th>
<th>A</th>
<th>R</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>654</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

ROT ← R
CR0 ← 0 || MSRpg || 0

if MSRpg = 0b00 then #Non-transactional
    TEXASR ← 0x000000000 || 0b0 || ROT || 0b0 ||
0x0000001
    TFHAR ← CIA + 4
    TDOOMED ← 0
    MSRrot ← 0b10
    checkpoint area ← (checkpointed registers)
    if not ROT and the transaction succeeds then
        enforce_barrier(mbl1)
        enforce_barrier(mbls)
        enforce_barrier(mb1)
        enforce Barrier(mbss)
else if MSRpg = 0b10 then #Transactional
    if TEXASROTL=TLmax then
        cause ← 0x014000000
        TMRecordFailure(cause)
        TMHandleFailure()
    else
        TEXASROTL ← TEXASROTL + 1
        if (TEXASROTR=1) & (not ROT)
        & the transaction succeeds
            enforce_barrier(mbl1)
            enforce_barrier(mbls)
            enforce_barrier(mb1)
            enforce Barrier(mbss)
        TEXASRROTr ← 0
        (Floating-point Storage Access instructions are not affected.)

The tbegin. instruction initiates execution of a transaction, either an outer transaction or a nested transaction, as described below.

An outer transaction is initiated when tbegin. is executed in the Non-transactional state. If R=0 and the transaction is successful, a memory barrier is inserted equivalent to that produced by a sync instruction with E=0b1111. (See Section 4.4.3.) TEXASR and TFHAR are initialized, and the TDOOMED bit is set to 0. A nested transaction is initiated when tbegin. is executed in the Transactional state unless the transaction level is already at its maximum value, in which case failure recording is performed with a failure cause of 0x014000000 and failure handling is performed. When initiating a nested transaction, the transaction level held in TEXASR is incremented by 1, and if TEXASR ROT =1 but R=0, and the transaction succeeds, a memory barrier is inserted equivalent to that produced by a sync instruction with E=0b1111 and TEXASR ROT is turned off. The effects of a nested transaction will not be visible until the outer transaction commits, and in the event of failure, the checkpointed registers are reverted to the pre-transactional values of the outer transaction. Initiation of a transaction is unsuccessful when in the Suspended state.

When successfully initiated, transactional execution continues until the transaction is terminated using a tend., tabort., tabortdc., tabortwci., tabortwc., or treclaim. instruction, suspended using a tsr instruction, or failure occurs. Upon transaction failure while in the Transactional state, transaction failure recording and failure handling are performed as defined in Section 5.3. Upon transaction failure while in the Suspended state, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred.
CR0 is set as follows.

<table>
<thead>
<tr>
<th>CR0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Transaction initiation successful, unnested</td>
</tr>
<tr>
<td></td>
<td>(Transaction state of Non-transactional prior to</td>
</tr>
<tr>
<td></td>
<td>tbegin.)</td>
</tr>
<tr>
<td>010</td>
<td>Transaction initiation successful, nested</td>
</tr>
<tr>
<td></td>
<td>(Transaction state of Transactional prior to</td>
</tr>
<tr>
<td></td>
<td>tbegin.)</td>
</tr>
<tr>
<td>001</td>
<td>Transaction initiation unsuccessful,</td>
</tr>
<tr>
<td></td>
<td>(Transaction state of Suspended prior to</td>
</tr>
<tr>
<td></td>
<td>tbegin.)</td>
</tr>
</tbody>
</table>

Other than the setting of CR0, tbegin. in the Suspended state is treated as a no-op.

The use of the A field is implementation specific.

Special Registers Altered
CR0 TEXASR TFHAR TS

Programming Note
When a transaction is successfully initiated, and failure subsequently occurs, control flow will be redirected to the instruction following the tbegin. instruction. When failure handling occurs, as described in Section 5.3.3, CR0 is set to 0b101 0. Consequently, instructions following tbegin. should also expect this value as an indication of transaction failure. Most applications will follow tbegin. with a conditional branch predicates on CR02; code at this target is responsible for handling the transaction failure. 

**Transaction End X-form**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The A=0 variant of tend. supports nested transactions, in which the transaction is committed only if the execution of tend. completes an outer transaction. Execution of this variant by a nested transaction (TEXASR_TL > 1) causes TEXASR_TL to be decremented by 1. The A=1 variant of tend. unconditionally completes the current outer transaction and all nested transactions.

When the tend. instruction completes an outer transaction, transaction commit is predicated on the TDOOMED bit. If TDOOMED is 1, failure handling occurs as defined in Section 5.3.3. If TDOOMED is 0, the transaction is committed, and TEXASR_TL is set to 0. In both cases, the transaction state is set to Non-transactional.

When the tend. instruction commits a transaction, it atomically commits its writes to storage. If TEXASR_ROT=0, the integrated cumulative memory barrier is inserted prior to the creation of the aggregate store, and a memory barrier is inserted equivalent to that produced by a sync instruction with E=0b1111 after the aggregate store. (See Section 4.4.3.) If the transaction has failed prior to the execution of tend., no storage updates are performed and no memory barrier is inserted. In either case (success or failure), all resources associated with the transaction are discarded.

If the transaction succeeds, Condition Register field 0 is set to 0 || MSR_TS || 0. If the transaction fails, CR0 is set to 0b101 0.
Other than the setting of CR0, `tend`, in Non-transactional state is treated as a no-op. If an attempt is made to execute `tend` in Suspended state, a TM Bad Thing type Program interrupt occurs.

### Special Registers Altered

CR0 TEXASR TS

### Extended Mnemonics

Extended mnemonics for transaction end.

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent To:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tend</code></td>
<td><code>tend. 0</code></td>
</tr>
<tr>
<td><code>tendall</code></td>
<td><code>tend. 1</code></td>
</tr>
</tbody>
</table>

### Programming Note

When an outer `tend` or a `tend` with `A=1` is executed in the Transactional state, the CR0 value 0b101 || 0 will never be visible to the instruction that immediately follows `tend`, because in the event of failure the failure handler will have been invoked not later than the completion of the `tend` instruction.

### Transaction Abort X-form

<table>
<thead>
<tr>
<th>tabort. RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 // 6 // 16 21 910 1</td>
</tr>
</tbody>
</table>

| CR0 ← 0 || MSR_PARAM_1 || 0 |

\[
\text{if } \text{MSR}_{\text{PS}} = 0b10 \lor \text{MSR[param] } = 0b01 \text{ then}
\]

\[
\begin{align*}
\text{#Transactional, or Suspended} & \\
\text{if } RA = 0 \text{ then cause } & \rightarrow \text{GPR(RA)}_{56:63} \lor 0x000001 \\
\text{else} & \text{cause } \rightarrow \text{GPR(RA)}_{56:63} \lor 0x000001 \\
\text{if } \text{MSR}_{\text{PS}} = 0b01 \land \text{TEXASR}_{\text{PS}} = 0 \text{ then } & \#\text{Suspended} \\
\text{Discard the transactional footprint} & \\
\text{TMRecordFailure(cause)} & \\
\text{if } \text{MSR}_{\text{PS}} = 0b10 \text{ then} & \#\text{Transactional} \\
\text{TMHandleFailure()} & \\
\end{align*}
\]

The `tabort`. instruction sets condition register field 0 to 0 || MSR_PARAM_1 || 0. When in the Transactional state or the Suspended state the `tabort`. instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to GPR(RA)_{56:63} || 0x000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3(this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of `tabort`. in the Non-transactional state is treated as a no-op.

### Special Registers Altered

CR0 TEXASR TFIAR TS
Transaction Abort Word Conditional

**X-form**

**tabortwc.**  

<table>
<thead>
<tr>
<th>TO</th>
<th>RA</th>
<th>RB</th>
<th>782</th>
<th>1</th>
</tr>
</thead>
</table>

\[
a \leftarrow \text{EXTS}(\text{RA}_{32:63})
\]

\[
b \leftarrow \text{EXTS}(\text{RB}_{32:63})
\]

\[
\text{abort} \leftarrow 0
\]

\[
\text{CR0} \leftarrow 0 || \text{MSR}_{TS} || 0
\]

- If \((a < b) \land \text{TO}_0\) then abort \(\leftarrow 1\)
- If \((a > b) \land \text{TO}_1\) then abort \(\leftarrow 1\)
- If \((a = b) \land \text{TO}_2\) then abort \(\leftarrow 1\)
- If \((a \neq b) \land \text{TO}_3\) then abort \(\leftarrow 1\)
- If abort \& \((\text{MSR}_{TS} = 0b10 \lor \text{MSR}_{TS} = 0b01)\) then
  - #Transactional or Suspended
    - cause \(\leftarrow 0x00000001\)
    - if \(\text{MSR}_{TS} = 0b01 \land \text{TEXASR}_{TS} = 0\) then #Suspended
      - discard transactional footprint
    - TMRecordFailure(cause)
    - if MSR_{TS} = 0b10 then #Transactional
      - TMHandleFailure()

The **tabortwc.** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA_{32:63} are compared with the contents of register RB_{32:63}. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the **tabortwc.** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of **tabortwc.** in the Non-transactional state is treated as a no-op.

**Special Registers Altered**

CR0 TEXASR TFIAR TS

---

Transaction Abort Word Conditional

**Immediate X-form**

**tabortwci.**  

<table>
<thead>
<tr>
<th>TO</th>
<th>RA</th>
<th>SI</th>
<th>846</th>
<th>1</th>
</tr>
</thead>
</table>

\[
a \leftarrow \text{EXTS}(\text{RA}_{32:63})
\]

\[
\text{abort} \leftarrow 0
\]

\[
\text{CR0} \leftarrow 0 || \text{MSR}_{TS} || 0
\]

- If \((a < \text{EXTS(SI)} \land \text{TO}_0)\) then abort \(\leftarrow 1\)
- If \((a > \text{EXTS(SI)} \land \text{TO}_1)\) then abort \(\leftarrow 1\)
- If \((a = \text{EXTS(SI)} \land \text{TO}_2)\) then abort \(\leftarrow 1\)
- If \((a \neq \text{EXTS(SI)} \land \text{TO}_3)\) then abort \(\leftarrow 1\)
- If abort \& \((\text{MSR}_{TS} = 0b10 \lor \text{MSR}_{TS} = 0b01)\) then
  - #Transactional or Suspended
    - cause \(\leftarrow 0x00000001\)
    - if \(\text{MSR}_{TS} = 0b01 \land \text{TEXASR}_{TS} = 0\) then #Suspended
      - discard transactional footprint
    - TMRecordFailure(cause)
    - if \(\text{MSR}_{TS} = 0b10\) then #Transactional
      - TMHandleFailure()

The **tabortwci.** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA_{32:63} are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the **tabortwci.** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of **tabortwci.** in the Non-transactional state is treated as a no-op.

**Special Registers Altered**

CR0 TEXASR TFIAR TS

---

Transaction Abort Doubleword Conditional

**X-form**

**tabortdc.**  

| TO | RA | RB | 814 | 1 |

\[
\text{tabortdc.} \leftarrow \text{TO,RA,RB}
\]
The `tabortdc` instruction sets condition register field 0 to 0 || MSR_T8 || 0. The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the `tabortdc` instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of `tabortdc` in the Non-transactional state is treated as a no-op.

### Special Registers Altered

- CR0: MSR_T8, TFIAR, TS

### Transaction Suspend or Resume X-form

The `tsr` instruction suspends or resumes a transaction based on the value of the L field in the special register TFIAR.

The `tsr` instruction sets condition register field 0 to 0 || MSR_T8 || 0. The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the `tabortdc` instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of `tabortdc` in the Non-transactional state is treated as a no-op.
The `tsr` instruction sets condition register field 0 to 0 || MSR₇₈ || 0. Based on the value of the L field, two variants of `tsr` are used to change the transaction state.

If L = 0, and the transaction state is Transactional, the transaction state is set to Suspended.

If L = 1, and the transaction state is Suspended, the transaction state is set to Transactional.

Other than the setting of CR0, the execution of `tsr` in the Non-transactional state is treated as a no-op.

### Special Registers Altered

CR0 TS

---

### Programming Note

When resuming a transaction that has encountered failure while in the Suspended state, failure handling is performed after the execution of `resume` and no later than the next failure synchronizing event.

### Extended Mnemonics

Extended mnemonics for Transaction Suspend or Resume.

<table>
<thead>
<tr>
<th>Extended</th>
<th>Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tsuspend</code></td>
<td><code>tsr</code> 0</td>
</tr>
<tr>
<td><code>tresume</code></td>
<td><code>tsr</code> 1</td>
</tr>
</tbody>
</table>

### Transaction Check X-form

<table>
<thead>
<tr>
<th>tcheck</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 BF // // // 718 /</td>
<td></td>
</tr>
</tbody>
</table>

If MSR₇₈ = 0b10 | MSR₇₈ = 0b01 then #Transaction
| for Suspended |
| for each load caused by an instruction following the outer `tbegin` and preceding this `tcheck` |
| if (Load instruction was executed in T state with TEXASR₉₀=0 or accessing a location previously stored transactionally) |
| (Load instruction was executed in S state with TEXASR₉₀=0 and accessed a location previously accessed transactionally) |
| (Load instruction was executed in S state with TEXASR₉₀=0 and accessing a location previously stored transactionally) |
| then wait until load has been performed with respect to all processors and mechanisms |
| CR field BF ← TDOOMED || MSR₇₈ || 0 |

If the transaction state is Transactional or Suspended, the `tcheck` instruction ensures that all loads that are caused by instructions that follow the outer `tbegin` instruction and precede the `tcheck` instruction and satisfy one of the following properties, have been performed with respect to all processors and mechanisms.

- The load is caused by an instruction that was executed in Transactional state, either while TEXASR₉₀=0 or accessing a location previously stored transactionally.
- The load is caused by an instruction that was executed in Suspended state while TEXASR₉₀=0 and accesses a location that was accessed transactionally.
- The load is caused by an instruction that was executed in Suspended state while TEXASR₉₀=1 and accesses a location that was stored transactionally.

The `tcheck` instruction then copies the TDOOMED bit into bit 0 of CR field BF, copies MSR₇₈ to bits 1:2 of CR field BF, and sets bit 3 of CR field BF to 0.

Other than the setting of CR field BF, execution of `tcheck` in the Non-transactional state is treated as a no-op.

### Special Registers Altered

CR field BF
One use of the `tcheck` instruction in Suspended state is to determine whether preceding loads from transactionally modified locations have returned the data the transaction stored. (If the transaction has failed, some of the loads may have returned a more recent value that was stored by a conflicting store, or may have returned the pre-transaction contents of the location.). It is important to use `tcheck` between any Suspended state loads that might access transactionally modified locations and subsequent computation using the Suspended-state-loaded data. Otherwise, corrupt data could cause problems such as wild branches or infinite loops.

Another use of `tcheck` in Suspended state is to determine whether the contents of storage, as seen in Suspended state, are consistent with the transaction succeeding -- e.g., whether no location that has been accessed transactionally (stored transactionally, for ROTs), and has been seen in Suspended state, has been subject to a conflict thus far. (A location is seen in Suspended state either by being loaded in Suspended state or by being loaded in Transactional state and the value (or a value derived therefrom) passed, in a register, into Suspended state.)

A use of `tcheck` in Transactional state is to determine whether the transaction still has the potential to succeed.

Note that `tcheck` provides an instantaneous check on the integrity of a subset of the accesses performed within a transaction. `tcheck` is not a failure synchronizing mechanism. Even if no accesses follow the `tcheck`, there may still be latent failures that haven’t been recorded, for example caused by accesses that `tcheck` does not wait for, by external conflicts that will happen in the future, or simply by time of flight to the failure detection mechanism for operations that have already been performed.

The `tcheck` instruction can return 1 in bit 0 of CR field BF before the failure has been recorded in TEXASR and TFIAR.

The `tcheck` instruction may cause pipeline synchronization. As a result, programs that use `tcheck` excessively may perform poorly.
Chapter 6.  Time Base

6.1 Time Base Overview

The time base facilities include a Time Base and an Alternate Time Base which is category: Alternate Time Base. The Alternate Time Base is analogous to the Time Base except that it may count at a different frequency and is not writable.

6.2 Time Base

The Time Base (TB) is a 64-bit register (see Figure 9) containing a 64-bit unsigned integer that is incremented periodically as described below.

The suggested frequency at which the time base increments is 512 MHz, however, variation from this rate is allowed provided the following requirements are met.

- The contents of the Time Base differ by no more than +/- four counts from what they would be if they incremented at the required frequency.
- Bit 63 of the Time Base is set to 1 between 30% and 70% of the time over any time interval of at least 16 counts.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock. The Time Base update frequency is not required to be constant. What is required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

6.2.1 Time Base Instructions

**Move From Time Base**  
XFX-form

```
mtfb  RT,TBR
```

[Category: Phased-Out]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBU</td>
<td>Upper 32 bits of Time Base</td>
</tr>
<tr>
<td>TBL</td>
<td>Lower 32 bits of Time Base</td>
</tr>
</tbody>
</table>

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from $2^{64}-1$ to $0$). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

Programming Note

If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

This instruction behaves as if it were an `mfspr` instruction; see the `mfspr` instruction description in Section 3.3.16 of Book I.

Special Registers Altered:

None
Extended Mnemonics:

Extended mnemonics for Move From Time Base:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mftb Rx</td>
<td>mftb Rx,268</td>
</tr>
<tr>
<td></td>
<td>mfspr Rx,268</td>
</tr>
<tr>
<td>mftbu Rx</td>
<td>mftb Rx,269</td>
</tr>
<tr>
<td></td>
<td>mfspr Rx,269</td>
</tr>
</tbody>
</table>

**Programming Note**

New programs should use `mfspr` instead of `mftb` to access the Time Base.

**Programming Note**

`mftb` serves as both a basic and an extended mnemonic. The Assembler will recognize an `mftb` mnemonic with two operands as the basic form, and an `mftb` mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

**Programming Note**

The `mfspr` instruction can be used to read the Time Base on all processors that comply with Version 2.01 of the architecture or any subsequent version.

It is believed that the `mfspr` instruction can be used to read the Time Base on most processors that comply with versions of the architecture that precede Version 2.01. Processors for which `mfspr` cannot be used to read the Time Base include the following.

- 601
- POWER3

(601 implements neither the Time Base nor `mftb`, but depends on software using `mftb` to read the Time Base, so that the attempt causes the Illegal Instruction error handler to be invoked and thereby permits the operating system to emulate the Time Base.)
Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base increments at the constant rate of 512 MHz. (Note, however, that programs should allow for the possibility that some implementations may not increment the least-significant 4 bits of the Time Base at a constant rate.) What is wanted is the pair of 32-bit values comprising a POSIX standard clock: the number of whole seconds that have passed since 00:00:00 January 1, 1970, UTC, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant `ticks_per_sec` contains the value 512,000,000, which is the number of times the Time Base is updated each second.
- The integer constant `ns_adj` contains the value \( \frac{1,000,000,000}{512,000,000} \times 2^{32} / 2 = 4194304000 \)
  which is the number of nanoseconds per tick of the Time Base, multiplied by \( 2^{32} \) for use in `mulhwu` (see below), and then divided by 2 in order to fit, as an unsigned integer, into 32 bits.

When the processor is in 64-bit mode, The POSIX clock can be computed with an instruction sequence such as this:

```assembly
mulld Rz,Rz,Rx  # Rz = quotient * divisor
sub Rz,Ry,Rz    # Rz = excess ticks
lwz  Rx,ns_adj
slwi Rz,Rz,1    # Rz = 2 * excess ticks
mulhwu Rz,Rz,Rx # mul by (ns/tick)/2 * 2^{32}
stw  Rz,posix_ns# product[0:31] = excess ns
```

For the Embedded environment when the processor is in 32-bit mode, it is not possible to read the Time Base using a single instruction. Instead, two instructions must be used, one of which reads TBL and the other of which reads TBU. Because of the possibility of a carry from TBL to TBU occurring between the two reads, a sequence such as the following must be used to read the Time Base.

```assembly
loop:
mfspr Rx,TBU  # load from TBU
mfspr Ry,TB  # load from TB
mfspr Rz,TBU  # load from TBU
cmp cr0,0,Rx,Rz# check if ‘old‘=‘new’
bne loop  #branch if carry occurred
```

### Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt (see Book III), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of `ticks_per_sec` for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute Time of Day use the current Time Base Value and the saved value.

---

6.3 Alternate Time Base [Category: Alternate Time Base]

The Alternate Time Base (ATB) is a 64-bit register (see Figure 9) containing a 64-bit unsigned integer that is incremented periodically. The frequency at which the integer is updated is implementation-dependent.

The ATBL register is an aliased name for the ATB.

The Alternate Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF \((2^{64} - 1)\). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no explicit indication (such as an interrupt; see Book III) that this has occurred.

The Alternate Time Base is accessible in both user and supervisor mode. The counter can be read by executing a `mfspr` instruction specifying the ATB (or ATBL) register, but cannot be written. A second SPR register ATBU, is defined that accesses only the upper 32 bits of the counter. Thus the upper 32 bits of the counter may be read into a register by reading the ATBU register.

The effect of entering a power-savings mode or of processor frequency changes on counting in the Alternate Time Base is implementation-dependent.
Chapter 7. Event-Based Branch Facility [Category: Server]

7.1 Event-Based Branch Overview

The Event-Based Branch facility allows application programs to enable hardware to change the effective address of the next instruction to be executed when certain events occur to an effective address specified by the program.

The operation of the Event-Based Branch facility is summarized as follows:

- The Event-Based Branch facility is available only when the system program has made it available. See Section 9.5 of Book III-S for additional information.

- When the Event-Based Branch facility is available, event-based branches are caused by event-based exceptions. Event-based exceptions can be enabled to occur by setting bits in the Event Control field of the BESCR.

- When an event-based exception occurs, the bit in the BESCR control field corresponding to the event-based exception is set to 0 and the bit in the Event Status field in the BESCR corresponding to the event-based exception is set to 1.

- If the global enable bit in the BESCR is set to 1 when any of the bits in the status field are set to 1 (i.e. when an event-based exception exists), an event-based branch occurs.

- The event-based branch causes the global enable bit to be set to 0, causes instruction fetch and execution to continue at the effective address contained in the EBBHR, and causes the TS field of the BESCR to indicate the transactional state of the processor when the event-based branch occurred. If the processor was in transactional state when the event-based branch occurred, it is put into suspended state. The EBBRR is set to the effective address of the instruction that would have attempted to execute next if no event-based branch had occurred.

- The event-based branch handler performs the necessary processing in response to the event, and then executes an `rfebb` instruction in order to resume execution at the instruction that would have been executed next when the event-based branch occurred. The `rfebb` instruction also restores the processor to the transactional state indicated by BESCRTS.

Additional information about the Event-Based Branch facility is given in Section 3.4 of Book III-S.

**Programming Note**

Since system software controls the availability of the Event-Based Branch facility (see Section 9.5 of Book III-S), an interface must be provided that enables applications to request access to the facility and determine when it is available.

**Programming Note**

In order to initialize the Event-Based Branch facility for Performance Monitor event-based exceptions, software performs the following operations.

- Software requests control of the Event-Based Branch facility from the system program.

- Software requests the System Program to initialize the Performance Monitor as desired.

- Software sets the EBBHR to the effective address of the event-based branch handler.

- Software enables Performance Monitor event-based exceptions in the BESCR Control Field by setting BESCRPME to 1. BESCRPME0 should also be set to 0.

- Software sets the GE bit in the BESCR to enable event-based branches.
7.2 Event-Based Branch Registers

7.2.1 Branch Event Status and Control Register

The Branch Event Status and Control Register (BESCR) is a 64-bit register that contains control and status information about the Event-Based Branch facility.

The entire BESCR can be read or written using SPR 806. Individual bits of the BESCR can be set or reset using two sets of additional SPR numbers.

- When mtspr indicates SPR 800 (Branch Event Status and Control Set, or BESCRS), the bits in BESCR which correspond to “1” bits in the source register are set to 1; all other bits in the BESCR are unaffected. SPR 801 (BESCRSU) provides the same capability to each of the upper 32 bits of the BESCR.

- When mtspr indicates SPR 802 (Branch Event Status and Control Reset, or BESCCR), the bits in BESCR which correspond to “1” bits in the source register are set to 0; all other bits in the BESCR are unaffected. SPR 803 (BESCRRU) provides the same capability to each of the upper 32 bits of the BESCR.

When mtspr indicates any of the above SPR numbers, the current value of the register is returned.

Programming Note

Event-based branch handlers typically reset event status bits upon entry, and enable event enable bits after processing an event. Execution of rfebb then re-enables the GE bit so that additional event-based branches can occur.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Global Enable (GE)</td>
</tr>
<tr>
<td>0</td>
<td>Event-based branches are disabled</td>
</tr>
<tr>
<td>1</td>
<td>Event-based branches are enabled.</td>
</tr>
</tbody>
</table>

When an event-based branch occurs, GE is set to 0 and is not altered by hardware until rfebb 1 is executed or software sets GE=1 and another event-based branch occurs.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:31</td>
<td>Event Control</td>
</tr>
<tr>
<td>1:30</td>
<td>Reserved</td>
</tr>
<tr>
<td>31</td>
<td>Performance Monitor Event-Based Exception Enable (PME)</td>
</tr>
<tr>
<td>0</td>
<td>Performance Monitor event-based exceptions are disabled.</td>
</tr>
<tr>
<td>1</td>
<td>Performance Monitor event-based exceptions are enabled until a Performance Monitor event-based exception occurs, at which time:</td>
</tr>
<tr>
<td></td>
<td>- PME is set to 0</td>
</tr>
<tr>
<td></td>
<td>- PMEO is set to 1</td>
</tr>
</tbody>
</table>

See Chapter 9 of Book III-S for information about performance monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:33</td>
<td>Transactional State [Category:TM]</td>
</tr>
<tr>
<td>00</td>
<td>Non-transactional</td>
</tr>
<tr>
<td>01</td>
<td>Suspended</td>
</tr>
<tr>
<td>10</td>
<td>Transactional</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Programming Note

Event-based branch handlers should not modify this field since its value is used by the processor to determine the transactional state of the processor after the rfebb instruction is executed.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34:62</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Figure 11. Branch Event Status and Control Register (BESCR)

<table>
<thead>
<tr>
<th>GE</th>
<th>Event Control</th>
<th>TS</th>
<th>Event Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>32</td>
<td>34</td>
</tr>
</tbody>
</table>

Figure 12. Branch Event Status and Control Register Upper (BESCRU)

<table>
<thead>
<tr>
<th>GE</th>
<th>Event Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

63  Performance Monitor Event-Based Exception Occurred (PMEO)
0  Performance Monitor event-based exception has not occurred since the last time software set this bit to 0.
1  A Performance Monitor event-based exception has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor event-based exception occurs. This bit can be set to 0 only by the mtspri instruction.

See Chapter 9 of Book III-S for information about performance monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

---

**Programming Note**

Software should set this bit to 0 after handling an event-based branch due to a Performance Monitor event-based exception.

---

7.2.2 Event-Based Branch Handler Register

The Event-Based Branch Handler Register (EBBHR) is a 64-bit register register that contains the 62 most significant bits of the effective address of the instruction that is executed next after an event-based branch occurs. Bits 62:32 must be available to be read and written by software.

<table>
<thead>
<tr>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>62 63</td>
</tr>
</tbody>
</table>

Figure 13. Event-Based Branch Handler Register (EBBHR)

---

7.2.3 Event-Based Branch Return Register

The Event-Based Branch Return Register (EBBRR) is a 64-bit register that contains the 62 most significant bits of an instruction effective address as specified below.

<table>
<thead>
<tr>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>62 63</td>
</tr>
</tbody>
</table>

---

**Figure 14. Event-Based Branch Return Register (EBBRR)**

When an event-based branch occurs, bits 0:61 of the EBBRR are set to the effective address of the instruction that the processor would have attempted to execute next if no event-based branch had occurred. Bits 62:63 are reserved.
7.3 Event-Based Branch Instructions

*Return from Event-Based Branch*  
**XL-form**

```
rfebb S
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

BESCRGE $\leftarrow$ S  
MSR$_{TS}$ $\leftarrow$ BESCR$_{FS}$  
NIA $\leftarrow$ \texttt{EBBRR0:61} || 0b00

BESCRGE is set to S. The processor is placed in the transactional state indicated by BESCR$_{TS}$.

If there are no pending event-based exceptions, then the next instruction is fetched from the address EBBRR$_{0:61}$ || 0b00 (when MSR$_{SF}=1$) or EBBRR$_{32:61}$ || 0b00 (when MSR$_{SF}=0$). If one or more pending event-based exceptions exist, an event-based branch is generated; in this case the value placed into EBBRR by the Event-Based Branch facility is the address of the instruction that would have been executed next had the event-based branch not occurred.

See Section 3.4 of Book III-S for additional information about this instruction.

**Special Registers Altered:**

BESCR  
MSR (See Book III-S)

**Extended Mnemonics:**

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfebb</td>
<td>rfebb 1</td>
</tr>
</tbody>
</table>

**Programming Note**

*rfebb* serves as both a basic and an extended mnemonic. The Assembler will recognize an *rfebb* mnemonic with one operand as the basic form, and an *rfebb* mnemonic with no operand as the extended form. In the extended form, the S operand is omitted and assumed to be 1.

If the BESCR$_{TS}$ has been modified by software after an event-based branch occurs, an illegal transaction state transition may occur. See Chapter 3.2.2 of Book III-S.

---

When an event-based branch occurs, the event-based branch handler executes the following sequence of operations. This sequence of operations assumes that the handler has access to a stack or other area in memory in which state information from the main program can be stored. Note also that in this example, the handler entry point is labeled “E,” r1 is used as a scratch register, and only Performance Monitor events are enabled.

```
E:Save state  // This is the entry pt
mfspr r1, BESCR // Check event status
Process event
r1 $\leftarrow$ 0x0000 0000 0000 0001
mtspr BESCR, r1
    //Reset PMAO event status bit
    //Note: The PMAO bit of MMCR0 must also
    //      be reset. See Book III-S.
     r1 $\leftarrow$ 0x0000 0001 0000 0000
mtspr BESCR$_{S}$, r1
    //Enable PME bit
    //Note: The PME bit of MMCR0 must also
    //      be enabled. See Book III-S.
Restore state
rfebb 1       // return & global enable
```
Chapter 8. Decorated Storage Facility [Category: Decorated Storage]

The Decorated Storage facility provides Load, Store, and Notify operations to storage that have additional semantics other than the reading and writing of data values to the addressed storage locations. A decoration is specified that provides semantic information about how the operation is to be performed. A decorated device is a device that implements an address range of storage, and applies decorations to operations performed on the address range of storage.

A Decorated Storage instruction specifies the following attributes:
- The type of access, which is either a Decorated Load, Decorated Store, or a Decorated Notify.
- The EA in register RB, to which the operation is to be performed.
- The decoration in register RA, which further defines what operation should be performed by the decorated device.
- The data itself, either data provided by the processor to the decorated device (in the case of a Decorated Load), or the data provided by the decorated device to be consumed by the processor (in the case of a Decorated Store). Decorated Notify operations do not contain data.

The semantics of any Decorated Storage operation that is Caching Inhibited are defined by the decorated device depending on whether it is a Decorated Load, Decorated Store, or Decorated Notify, and the value supplied as a decoration. Such semantics may differ from decorated device to decorated device similar to how devices other than well-behaved memory may treat Load and Store operations. The semantics of any operation associated with a Decorated Storage operation that is not Caching Inhibited are the same as an analogous Load or Store instruction of the same data size.

The results of a Decorated Storage operation that is Caching Inhibited to a device that does not support decorations is boundedly undefined. The results of a Load or Store operation that is Caching Inhibited to a decorated device that requires a decoration is boundedly undefined.

For Decorated Load operations, a Load operation with the specified decoration is performed to the EA and the data provided by the decorated device is placed in the target register.

For Decorated Store operations, a Store operation using the data specified in the source register with the specified decoration is performed to the EA.

Decorated Load instructions are treated as Load instructions for address translation, access control, debug events, storage attributes, alignment, and memory access ordering. Decorated Store instructions are treated as Store instructions for address translation, access control, debug events, storage attributes, alignment, and memory access ordering. A Decorated Notify instruction is treated as a zero byte Store for address translation, access control, debug events, storage attributes, alignment, and memory access ordering.

**Programming Note**
Software should be acutely aware of how transactions to a decorated device that implements Decorated Storage will occur. Not only does this imply knowing the particular decorated device’s semantics, but also ensuring that the transactions are appropriately issued by the processor. This includes alignment, speculative accesses, and ordering. In general, Caching Inhibited accesses are required to be Guarded and properly aligned.
8.1 Decorated Load Instructions

**Load Byte with Decoration Indexed X-form**

```plaintext
lbdx RT,RA,RB
```

EA \(\leftarrow (RB)\)

RT \(\leftarrow 560 \ || \ \text{MEM\_DECORATED}(EA,1,(RA))\)

Let the effective address (EA) be the contents of RB. The byte in storage addressed by EA is loaded using the decoration supplied by (RA) into RT\textsubscript{56:63}. RT\textsubscript{0:55} are set to 0.

**Special Registers Altered:**
None

**Load Halfword with Decoration Indexed X-form**

```plaintext
lhdx RT,RA,RB
```

EA \(\leftarrow (RB)\)

RT \(\leftarrow 480 \ || \ \text{MEM\_DECORATED}(EA,2,(RA))\)

Let the effective address (EA) be the contents of RB. The halfword in storage addressed by EA is loaded using the decoration supplied by (RA) into RT\textsubscript{48:63}. RT\textsubscript{0:47} are set to 0.

**Special Registers Altered:**
None

**Load Word with Decoration Indexed X-form**

```plaintext
lwdx RT,RA,RB
```

EA \(\leftarrow (RB)\)

RT \(\leftarrow 320 \ || \ \text{MEM\_DECORATED}(EA,4,(RA))\)

Let the effective address (EA) be the contents of RB. The word in storage addressed by EA is loaded using the decoration supplied by (RA) into RT\textsubscript{32:63}. RT\textsubscript{0:31} are set to 0.

**Special Registers Altered:**
None

**Load Doubleword with Decoration Indexed X-form**

```plaintext
lddx RT,RA,RB [Co-requisite category: 64-Bit]
```

EA \(\leftarrow (RB)\)

RT \(\leftarrow \text{MEM\_DECORATED}(EA,8,(RA))\)

Let the effective address (EA) be the contents of RB. The doubleword in storage addressed by EA is loaded using the decoration supplied by (RA) into RT.

**Special Registers Altered:**
None

**Load Floating Doubleword with Decoration Indexed X-form**

```plaintext
lfddx FRT,RA,RB [Co-requisite category: FP]
```

EA \(\leftarrow (RB)\)

FRT \(\leftarrow \text{MEM\_DECORATED}(EA,8,(RA))\)

Let the effective address (EA) be the contents of RB. The doubleword in storage addressed by EA is loaded using the decoration supplied by (RA) into FRT.

**Special Registers Altered:**
None
8.2 Decorated Store Instructions

**Store Byte with Decoration Indexed X-form**

```
stbdx RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>643</th>
<th>/</th>
</tr>
</thead>
</table>

EA ← (RB)
MEM_DECORATED(EA, 1, (RA)) ← (RS)_{56:63}

Let the effective address (EA) be the contents of RB.
(RS)_{56:63} are stored to the byte in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:
None

**Store Halfword with Decoration Indexed X-form**

```
sthdx RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>675</th>
<th>/</th>
</tr>
</thead>
</table>

EA ← (RB)
MEM_DECORATED(EA, 2, (RA)) ← (RS)_{48:63}

Let the effective address (EA) be the contents of RB.
(RS)_{48:63} are stored to the halfword in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:
None

**Store Word with Decoration Indexed X-form**

```
stwdx RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>707</th>
<th>/</th>
</tr>
</thead>
</table>

EA ← (RB)
MEM_DECORATED(EA, 4, (RA)) ← (RS)_{32:63}

Let the effective address (EA) be the contents of RB.
(RS)_{32:63} are stored to the word in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:
None

**Store Doubleword with Decoration Indexed X-form** [Co-requisite category: 64-Bit]

```
stddx RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
<th>739</th>
<th>/</th>
</tr>
</thead>
</table>

EA ← (RB)
MEM_DECORATED(EA, 8, (RA)) ← (RS)

Let the effective address (EA) be the contents of RB.
(RS) is stored to the doubleword in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:
None

**Store Floating Doubleword with Decoration Indexed X-form** [Co-requisite category: FP]

```
stfddx FRS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>FRS</th>
<th>RA</th>
<th>RB</th>
<th>931</th>
<th>/</th>
</tr>
</thead>
</table>

EA ← (RB)
MEM_DECORATED(EA, 8, (RA)) ← (FRS)

Let the effective address (EA) be the contents of RB.
(FRS) is stored to the doubleword in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:
None
8.3 Decorated Notify Instructions

Decorated Storage Notify X-form

dsn RA, RB

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>//</th>
<th>RA</th>
<th>RB</th>
<th>483</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

EA \(\rightarrow\) (RB)
MEM_NOTIFY(EA, (RA))

Let the effective address (EA) be the contents of RB. The decoration supplied by (RA) is sent to the address in storage specified by EA.

Special Registers Altered:

None
Chapter 9. External Control [Category: External Control]

The External Control category of facilities and instructions permits a program to communicate with a special-purpose device. Two instructions are provided, both of which must be implemented if the facility is provided.

- **External Control In Word Indexed** (*eciwx*), which does the following:
  - Computes an effective address (EA) like most X-form instructions
  - Validates the EA as would be done for a load from that address
  - Translates the EA to a real address
  - Transmits the real address to the device
  - Accepts a word of data from the device and places it into a General Purpose Register

- **External Control Out Word Indexed** (*ecowx*), which does the following:
  - Computes an effective address (EA) like most X-form instructions
  - Validates the EA as would be done for a store to that address
  - Translates the EA to a real address
  - Transmits the real address and a word of data from a General Purpose Register to the device

Permission to execute these instructions and identification of the target device are controlled by two fields, called the E bit and the RID field respectively. If attempt is made to execute either of these instructions when E=0 the system data storage error handler is invoked. The location of these fields is described in Book III.

The storage access caused by *eciwx* and *ecowx* is performed as though the specified storage location is Caching Inhibited and Guarded, and is neither Write Through Required nor Memory Coherence Required.

Interpretation of the real address transmitted by *eciwx* and *ecowx* and of the 32-bit value transmitted by *ecowx* is up to the target device, and is not specified by the Power ISA. See the System Architecture documentation for a given Power ISA system for details on how the External Control facility can be used with devices on that system.

**Example**

An example of a device designed to be used with the External Control facility might be a graphics adapter. The *ecowx* instruction might be used to send the device the translated real address of a buffer containing graphics data, and the word transmitted from the General Purpose Register might be control information that tells the adapter what operation to perform on the data in the buffer. The *eciwx* instruction might be used to load status information from the adapter.

A device designed to be used with the External Control facility may also recognize events that indicate that the address translation being used by the processor has changed. In this case the operating system need not “pin” the area of storage identified by an *eciwx* or *ecowx* instruction (i.e., need not protect it from being paged out).
9.1 External Access Instructions

In the instruction descriptions the statements “this instruction is treated as a Load” and “this instruction is treated as a Store” have the same meanings as for the Cache Management instructions; see Section 4.3.

External Control In Word Indexed X-form
eciwx RT,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
raddr ← address translation of EA
send load word request for raddr to
device identified by RID
RT ← 320 || word from device

Let the effective address (EA) be the sum (RA|0)+(RB).

A load word request for the real address corresponding to EA is sent to the device identified by RID, bypassing the cache. The word returned by the device is placed into RT32:63. RT0:31 are set to 0.

The E bit must be 1. If it is not, the data storage error handler is invoked.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

This instruction is treated as a Load.

See Book III-S for additional information about this instruction.

Special Registers Altered:
None

Programming Note

The eieio<SS> or mbar<E> instruction can be used to ensure that the storage accesses caused by eciwx and ecowx are performed in program order with respect to other Caching Inhibited and Guarded storage accesses.

External Control Out Word Indexed X-form
ecowx RS,RA,RB

<table>
<thead>
<tr>
<th></th>
<th>RS</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
raddr ← address translation of EA
send store word request for raddr to
device identified by RID
send (RS)32:63 to device

Let the effective address (EA) be the sum (RA|0)+(RB).

A store word request for the real address corresponding to EA and the contents of RS32:63 are sent to the device identified by RID, bypassing the cache.

The E bit must be 1. If it is not, the data storage error handler is invoked.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

This instruction is treated as a Store, except that its storage access is not performed in program order with respect to accesses to other Caching Inhibited and Guarded storage locations unless software explicitly imposes that order.

See Book III-S for additional information about this instruction.

Special Registers Altered:
None
Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book II. Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Data Cache Block Flush Mnemonics

The L field in the Data Cache Block Flush instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: dcbf serves as both a basic and an extended mnemonic. The Assembler will recognize a dcbf mnemonic with three operands as the basic form, and a dcbf mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

\[
\begin{align*}
dcbf RA,RB & \quad (\text{equivalent to: } dcbf RA,RB,0) \\
dcbfl RA,RB & \quad (\text{equivalent to: } dcbf RA,RB,1)
\end{align*}
\]

A.2 Load and Reserve Mnemonics

The EH field in the Load and Reserve instructions provides a hint regarding the type of algorithm implemented by the instruction sequence being executed. Extended mnemonics are provided that allow the EH value to be omitted and assumed to be 0b0.

Note: lbarx, lbarx, lwarx, ldarx, and lqarx serve as both basic and extended mnemonics. The Assembler will recognize these mnemonics with four operands as the basic form, and these mnemonics with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

\[
\begin{align*}
lbarx RT,RA,RB & \quad (\text{equivalent to: } lbarx RT,RA,RB,0) \\
lharx RT,RA,RB & \quad (\text{equivalent to: } lharx RT,RA,RB,0) \\
lwarx RT,RA,RB & \quad (\text{equivalent to: } lwarx RT,RA,RB,0) \\
ldarx RT,RA,RB & \quad (\text{equivalent to: } ldarx RT,RA,RB,0) \\
lqarx RT,RA,RB & \quad (\text{equivalent to: } lqarx RT,RA,RB,0)
\end{align*}
\]

A.3 Synchronize Mnemonics

The L field in the Synchronize instruction controls the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand. Two extended mnemonics are provided for the L=0 value in order to support Assemblers that do not recognize the sync mnemonic.

Note: sync serves as both a basic and an extended mnemonic. The Assembler will recognize a sync mnemonic with one operand as the basic form, and a sync mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

\[
\begin{align*}
sync & \quad (\text{equivalent to: } sync 0) \\
msync<E> & \quad (\text{equivalent to: } sync 0) \\
lwsync & \quad (\text{equivalent to: } sync 1) \\
ptesync<S> & \quad (\text{equivalent to: } sync 2)
\end{align*}
\]

A.4 Wait Mnemonics

The WC field in the wait instruction determines the condition that causes instruction execution to resume. Extended mnemonics are provided that represent the WC value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: wait serves as both a basic and an extended mnemonic. The Assembler will recognize a wait mnemonic with one operand as the basic form, and a wait mnemonic with no operands as the extended form. In the extended form the WC operand is omitted and assumed to be 0.

\[
\begin{align*}
wait & \quad (\text{equivalent to: } wait 0) \\
waitrsv & \quad (\text{equivalent to: } wait 1) \\
waitimpl & \quad (\text{equivalent to: } wait 2)
\end{align*}
\]
Appendix B. Programming Examples for Sharing Storage

This appendix gives examples of how dependencies and the Synchronization instructions can be used to control storage access ordering when storage is shared between programs.

Many of the examples use extended mnemonics (e.g., bne, bne-, cmpw) that are defined in Appendix E of Book I.

Many of the examples use the Load And Reserve and Store Conditional instructions, in a sequence that begins with a Load And Reserve instruction and ends with a Store Conditional instruction (specifying the same storage location as the Load Conditional) followed by a Branch Conditional instruction that tests whether the Store Conditional instruction succeeded.

In these examples it is assumed that contention for the shared resource is low; the conditional branches are optimized for this case by using “+” and “-” suffixes appropriately.

The examples deal with words; they can be used for doublewords by changing all word-specific mnemonics to the corresponding doubleword-specific mnemonics (e.g., lwarx to ldarx, cmpw to cmpd).

In this appendix it is assumed that all shared storage locations are in storage that is Memory Coherence Required, and that the storage locations specified by Load And Reserve and Store Conditional instructions are in storage that is neither Write Through Required nor Caching Inhibited.

B.1 Atomic Update Primitives

This section gives examples of how the Load And Reserve and Store Conditional instructions can be used to emulate atomic read/modify/write operations.

An atomic read/modify/write operation reads a storage location and writes its next value, which may be a function of its current value, all as a single atomic operation. The examples shown provide the effect of an atomic read/modify/write operation, but use several instructions rather than a single atomic instruction.

Fetch and No-op

The “Fetch and No-op” primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

loop:
  lwarx r4,0,r3  #load and reserve
  stwcx. r4,0,r3  #store old value if # still reserved
  bne-  loop  #loop if lost reservation

Note:

1. The stwcx, if it succeeds, stores to the target location the same value that was loaded by the preceding lwarx. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the lwarx is still the current value at the time the stwcx. is executed.

Fetch and Store

The “Fetch and Store” primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

loop:
  lwarx r4,0,r3  #load and reserve
  stwcx. r4,0,r3  #store new value if # still reserved
  bne-  loop  #loop if lost reservation
Fetch and Add

The “Fetch and Add” primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```assembly
loop:
  lwarx r5,0,r3 #load and reserve
  add r0,r4,r5 #increment word
  stwcx. r0,0,r3 #store new value if still res’ved
  bne- loop #loop if lost reservation
```

Fetch and AND

The “Fetch and AND” primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```assembly
loop:
  lwarx r5,0,r3 #load and reserve
  and r0,r4,r5 #AND word
  stwcx. r0,0,r3 #store new value if still res’ved
  bne- loop #loop if lost reservation
```

Test and Set

This version of the “Test and Set” primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (nonzero) is in GPR 4, and the old value is returned in GPR 5.

```assembly
loop:
  lwarx r5,0,r3 #load and reserve
  cmpwi r5,0 #done if word not equal to 0
  bne- exit #skip if not
  stwcx. r5,0,r3 #store new value if still res’ved
  bne- loop #loop if lost reservation
exit:
  mr r4,r6 #return value from storage
```

Compare and Swap

The “Compare and Swap” primitive atomically compares a value in a register with a word in storage. If they are equal stores the value from a second register into the word in storage, if they are unequal loads the word from storage into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR 3, the comparand is in GPR 4 and the old value is returned there, and the new value is in GPR 5.

```assembly
loop:
  lwarx r6,0,r3 #load and reserve
  cmpw r4,r6 #1st 2 operands equal?
  bne- exit #skip if not
  stwcx. r5,0,r3 #store new value if still res’ved
  bne- loop #loop if lost reservation
exit:
  mr r4,r6 #return value from storage
```

Notes:

1. The semantics given for “Compare and Swap” above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.

2. “Compare and Swap” is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by lwarx and stwcx. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.

3. In some applications the second bne- instruction and/or the mr instruction can be omitted. The bne- is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates “not equal” then (r4) and (r6) are in fact not equal. The mr is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.
B.2 Lock Acquisition and Release, and Related Techniques

This section gives examples of how dependencies and the Synchronization instructions can be used to implement locks, import and export barriers, and similar constructs.

B.2.1 Lock Acquisition and Import Barriers

An “import barrier” is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A sync instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

B.2.1.1 Acquire Lock and Import Shared Storage

If lwarx and stwcx instructions are used to obtain the lock, an import barrier can be constructed by placing an isync instruction immediately following the loop containing the lwarx and stwcx. The following example uses the “Compare and Swap” primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```assembly
loop:
lwarx r6,0,r3,1 #load lock and reserve
cmpw r4,r6    #skip ahead if
bne- wait     # lock not free
stwcx r5,0,r3 #try to set lock
bne- loop     #loop if lost reservation
isync         #import barrier
lwz r7,data1(r9)#load shared data
.
wait...        #wait for lock to free
```

The hint provided with lwarx indicates that after the program acquires the lock variable (i.e. stwcx is successful), it will release it (i.e. store to it) prior to another program attempting to modify it.

The second bne- does not complete until CR0 has been set by the stwcx. The stwcx does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the stwcx completes successfully. Together, the second bne- and the subsequent isync create an import barrier that prevents the load from “data1” from being performed until the branch has been resolved not to be taken.

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an lwsync instruction can be used instead of the isync instruction. If lwsync is used, the load from “data1” may be performed before the stwcx.. But if the stwcx. fails, the second branch is taken and the lwarx is re-executed. If the stwcx. succeeds, the value returned by the load from “data1” is valid even if the load is performed before the stwcx., because the lwsync ensures that the load is performed after the instance of the lwarx that created the reservation used by the successful stwcx..

B.2.1.2 Obtain Pointer and Import Shared Storage

If lwarx and stwcx instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the “Fetch and Add” primitive to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```assembly
loop:
lwarx r5,0,r3 #load pointer and reserve
add r0,r4,r5 #increment the pointer
stwcx r0,0,r3 #try to store new value
bne- loop   #loop if lost reservation
lwz r7,data1(r9) #load shared data
```

The load from “data1” cannot be performed until the pointer value has been loaded into GPR 5 by the lwarx. The load from “data1” may be performed before the stwcx.. But if the stwcx. fails, the branch is taken and the value returned by the load from “data1” is discarded. If the stwcx. succeeds, the value returned by the load from “data1” is valid even if the load is performed before the stwcx., because the load uses the pointer value returned by the instance of the lwarx that created the reservation used by the successful stwcx..

An isync instruction could be placed between the bne- and the subsequent lwz, but no isync is needed if all accesses to the shared data structure depend on the value returned by the lwarx.
B.2.2 Lock Release and Export Barriers

An "export barrier" is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor before the store that releases the lock is performed with respect to that processor.

B.2.2.1 Export Shared Storage and Release Lock

A \texttt{sync} instruction can be used as an export barrier independent of the storage control attributes (e.g., presence or absence of the Caching Inhibited attribute) of the storage containing the shared data structure. Because the lock must be in storage that is neither Write Through Required nor Caching Inhibited, if the shared data structure is in storage that is Write Through Required or Caching Inhibited a \texttt{sync} instruction must be used as the export barrier.

In this example it is assumed that the shared data structure is in storage that is Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

\begin{verbatim}
  stw r7, data1(r9) #store shared data (last)
  sync          #export barrier
  stw r4, lock(r3) #release lock
\end{verbatim}

The \texttt{sync} ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the \texttt{sync} have been performed with respect to that processor.

B.2.2.2 Export Shared Storage and Release Lock using lwsync

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, \texttt{lwsync} instruction can be used as the export barrier. Using \texttt{lwsync} rather than \texttt{sync} will yield better performance in most systems.

In this example it is assumed that the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

\begin{verbatim}
  stw r7, data1(r9) #store shared data (last)
  lwsync          #export barrier
  stw r4, lock(r3) #release lock
\end{verbatim}

The \texttt{lwsync} ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the \texttt{lwsync} have been performed with respect to that processor.

B.2.3 Safe Fetch

If a load must be performed before a subsequent store (e.g., the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the storage operand to be loaded is in GPR 3, the contents of the storage operand are returned in GPR 4, and the address of the storage operand to be stored is in GPR 5.

\begin{verbatim}
  lwz r4, 0(r3) #load shared data
  cmpw r4, r4  #set CR0 to 'equal'
  bne- $-8    #branch never taken
  stw r7, 0(r5) #store other shared data
\end{verbatim}

An alternative is to use a technique similar to that described in Section B.2.1.2, by causing the \texttt{stw} to depend on the value returned by the \texttt{lwz} and omitting the \texttt{cmpw and bne-}. The dependency could be created by ANDing the value returned by the \texttt{lwz} with zero and then adding the result to the value to be stored by the \texttt{stw}. If both storage operands are in storage that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the \texttt{cmpw and bne-} with an \texttt{lwsync} instruction.
Appendix B. Programming Examples for Sharing Storage

B.3 List Insertion

This section shows how the `lwarx` and `stwcx` instructions can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The “next element pointer” from the list element after which the new element is to be inserted, here called the “parent element”, is stored into the new element, so that the new element points to the next element in the list; this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR 3, the address of the new element is in GPR 4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:
lwarx r2,0,r3 #get next pointer
stw r2,0(r4)#store in new element
lwsync or sync #order stw before stwcx
stwcx. r4,0,r3 #add new element to list
bne- loop #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, “livelock” can occur. (Livelock is a state in which processors interact in a way such that no processor makes forward progress.)

If it is not possible to allocate list elements such that each element’s next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
lwz r2,0(r3)#get next pointer
loop1:
mr r5,r2 #keep a copy
stw r2,0(r4)#store in new element
sync #order stw before stwcx.
and before lwarx
lwarx r2,0,r3 #try again, reserving
cmpwi r2,0 #  (likely to succeed)
bne- loop
stwcx.r4,0,r3 #try to store non-0
bne- loop #loop if lost reserv’n
```

B.4 Notes

The following notes apply to Section B.1 through Section B.3.

1. To increase the likelihood that forward progress is made, it is important that looping on `lwarx/stwcx.` pairs be minimized. For example, in the “Test and Set” sequence shown in Section B.1, this is achieved by testing the old value before attempting the store; were the order reversed, more `stwcx.` instructions might be executed, and reservations might more often be lost between the `lwarx` and the `stwcx.`

2. The manner in which `lwarx` and `stwcx.` are communicated to other processors and mechanisms, and between levels of the storage hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a `lwarx` instruction that fails to return a desired value. For example, in the “Test and Set” sequence shown in Section B.1, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the “bne- exit” to “bne- loop”. However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows.

```
loop:
lwz r5,0(r3)#load the word
cmplwi r5,0 #loop back if word
bne- loop #  not equal to 0
lwarx r5,0,r3 #try again, reserving
lwarx r5,0 # (likely to succeed)
bne- loop
stwcx.r4,0,r3 #try to store non-0
bne- loop #loop if lost reserv’n
```

3. In a multiprocessor, livelock is possible if there is a `store` instruction (or any other instruction that can clear another processor’s reservation; see Section 1.7.3.1) between the `lwarx` and the `stwcx.` of a `lwarx/stwcx.` loop and any byte of the storage location specified by the Store is in the reservation granule. For example, the first code sequence shown in Section B.3 can cause livelock if two list elements have next element pointers in the same reservation granule.

B.5 Transactional Lock Elision

[Category: Transactional Memory]

This section illustrates the use of the Transactional Memory facility to implement transactional lock elision (TLE), in which lock-based critical sections are speculatedly executed as a transaction without first acquiring a lock. This locking protocol is an alternative to the rou-
tines described above, yielding increased concurrency when the lock that guards a critical section is frequently unnecessary.

### B.5.1 Enter Critical Section

The following example shows the entry point to a critical section using transactional lock elision. The entry code starts a transaction using the `tbegin` instruction and checks whether the transaction was aborted or not. If not, it checks whether the lock is free or not. If the lock is found to be free, the thread proceeds to execute the critical section.

In this example it is assumed that the address of the lock is in GPR 3, and the value indicating that the lock is free is in GPR 4. The handling of cases of transaction abort and busy lock are described in subsequent examples.

```assembly
tle_entry:
    tbegin.            #Start TLE transaction
    beq- tle_abort     #Handle TLE transaction abort
    lwz r6,0(r3)       #Read lock
    cmpw r6,r4         #Check if lock is free
    bne- busy_lock     #If not, handle lock busy case

    critical_section1:
```

### B.5.2 Handling Busy Lock

In the event that the lock is already held, by either another thread or the current thread, the transaction is aborted using the `tabort` instruction, using a software-defined code `TLE_BUSY_LOCK` indicating the cause of the abort. The abort returns control to the `beq` following `tbegin`, in the critical section entrance sequence, allowing for an abort handler to react appropriately.

```assembly
busy_lock:
    li r3, TLE_BUSY_LOCK
    tabort r3            #Abort TLE transaction
```

### B.5.3 Handling TLE Abort

A TLE transaction may fail for one of a variety of causes, persistent and transient. Persistent causes are certain—or at least highly likely—to cause future attempts to execute the same transaction to fail. However, for transient causes, it is possible that the failure cause may not be re-encountered in a subsequent attempt. Thus, persistent aborts are handled by taking a non-transactional path that involves the actual acquisition of the lock, while transient aborts retry the critical section using TLE.

The following example illustrates the handling of aborts in TLE. It is assumed that the address of the lock is in GPR 3. The immediate value of the `andis` instruction selects the Failure Persistent bit in the upper half of "TEXASR to be tested.

```assembly
tle_abort:
    mspr r4, TEXASRU  # Read high-order half
    # of "TEXASR
    andis. r5,r4,0x0100 # determine whether failure
    # is likely to be persistent
    bne tle_acquire_lock #Persistent, acquire lock
    #enter critical sec
    b tle_entry         #Transient, try TLE again
```

This example can be extended to keep track of the number of transient aborts and fall back on the acquisition of the lock after the number of transient failures reaches some threshold. It can also be extended to handle reentrant locks. Acquisition of TLE locks is described in a subsequent example.

### B.5.4 TLE Exit Section Critical Path

The following example illustrates the instruction sequence used to exit a TLE critical section. The CR0 value set by `tend` indicates whether the current thread was in a transaction. If so, the exited critical section was entered speculatively, and the transaction is ended. If not, the execution takes a path to release the lock.

Release of an acquired TLE lock is described in a subsequent example.

```assembly
tle_exit:
    tend.                   #End the current trans-
    #action, if any
    bng- tle_release_lock   #Release lock, if was
    #not in a transaction
```

### B.5.5 Acquisition and Release of TLE Locks

The steps for acquiring and releasing a lock associated with a TLE critical section are nearly identical to those for acquiring and releasing conventional locks that are not elided, as described in Section B.2. The only difference is special care must be taken to prevent loads and stores inside the critical section protected by the lock from being performed before the `Store Conditional` instruction to the lock variable.

The `isync` from the `Acquire Lock and Import Shared Storage` sequence described in Section B.2.1.1 is insufficient for ordering subsequent instructions. Instead, a
stronger storage access ordering instruction is needed. The following example shows the resulting steps for acquiring a TLE lock, replacing the `isync` instruction with a `sync` instruction. In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5.

tle_acquire_lock:
  lwax r10,0,r3,1       #Load lock and reserve
  cmpw r4,r10            #Skip ahead if
  bne- wait              #Lock not free
  stwcx. r5,0,r3         #Try to set lock
  bne- tle_acquire_lock  #Loop if lost reservation
  sync                   #Import barrier for TLE

critical_section1:

The instruction sequence necessary for the release of a TLE lock is identical to the conventional lock release sequence, where `lwsync` is sufficient to prevent the store that releases the lock from being performed before the loads and stores in the critical section protected by the lock.
Book III-S:

Power ISA Operating Environment Architecture - Server Environment [Category: Server]
Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Hypervisor Data Storage interrupt”, or “Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Hypervisor Emulation Assistance interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Hypervisor Instruction Storage interrupt”, or “Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt”.
- For “system trap handler” substitute “Trap type Program interrupt”.
- For “system Facility Unavailable error handler” substitute “Facility Unavailable interrupt” or “Hypervisor Facility Unavailable interrupt.”

1.2.1 Definitions and Notation

The definitions and notation given in Book I are augmented by the following.

- **Threaded processor, single-threaded processor, thread**
  A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. The only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor resources.

- **real page**
  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size is 4KB.

- **context of a program**
  The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR and SDR1, of certain lookaside buffers, such as the SLB and TLB, and of the Page Table.

- **exception**
  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

- **interrupt**
  The act of changing the machine state in response to an exception, as described in Chapter 6. “Interrupts” on page 939.
Additional exceptions to the sequential execution model, beyond those described in the section entitled “Instruction Fetching” in Book I, are the following.

- A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)

- A context-altering instruction is executed (Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.

- A Reference and Change bit is updated by the thread. The update need not be performed with respect to that thread until the required subsequent synchronizing operation has occurred.

- A Branch instruction is executed and the branch is taken. The update of the Come-From Address Register<sup>S</sup> (see Section 8.1.1 of Book III-S) need not occur until a subsequent context synchronizing operation has occurred.

“must”
If hypervisor software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage in real addressing mode, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc.

Hardware
Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.

Hypervisor privileged
A term used to describe an instruction or facility that is available only when the thread is in hypervisor state.

Privileged state and supervisor mode
Used interchangeably to refer to a state in which privileged facilities are available.

Problem state and user mode
Used interchangeably to refer to a state in which privileged facilities are not available.

/, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

?, ??, ???, ... denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

1.2.2 Reserved Fields

Book I’s description of the handling of reserved bits in System Registers, and of reserved values of defined fields of System Registers, applies also to the SLB. Book I’s description of the handling of reserved values of defined fields of System Registers applies also to architected storage tables (e.g., the Page Table).

Some fields of certain architected storage tables may be written to automatically by the hardware, e.g., Reference and Change bits in the Page Table. When the hardware writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.
- Contents of reserved fields are either preserved or written as zero.

Programming Note
Software should set reserved fields in the SLB and in architected storage tables to zero, because these fields may be assigned a meaning in some future version of the architecture.

1.3 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions that the processing unit can execute fall into the following classes:

- Instructions executed in the Branch Facility
- Instructions executed in the Fixed-Point Facility
- Instructions executed in the Floating-Point Facility
- Instructions executed in the Vector Facility

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book II may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged...
instructions for cache management). Instructions related to the privileged state, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

1.4 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSRFP=0 (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the thread is in privileged but non-hypervisor state (see Chapter 2), or an attempt to execute a hypervisor-only instruction (e.g., tlbie) when the thread is in privileged but non-hypervisor state
- the execution of a traced instruction (Trace interrupt)
- the execution of a Vector instruction when the vector facility is unavailable (Vector Unavailable interrupt)

1.5 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.5.1 Context Synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below. Such instructions and events are collectively called context synchronizing operations. The context synchronizing operations are the isync instruction, the System Linkage instructions, the mtmsrd instructions with L=0, and most interrupts (see Section 6.4).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.

2. The operation is not initiated or, in the case of isync, does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.

3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.

4. If the operation directly causes an interrupt (e.g., sc directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 6.8).

5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 5.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.5.2.

Unlike the Synchronize instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for isync (and sync and ptesync; see Section 1.5.2) because all other execution synchronizing operations also alter context.
1.5.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). `sync` and `ptesync` are treated like `isync` with respect to item 2. The execution synchronizing instructions are `sync`, `ptesync`, the `mtmsr[d]` instructions with L=1, and all context synchronizing instructions.

**Programming Note**

Unlike a context synchronizing operation, an execution synchronizing instruction does not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.
Chapter 2. Logical Partitioning (LPAR) and Thread Control

2.1 Overview

The Logical Partitioning (LPAR) facility permits threads and portions of real storage to be assigned to logical collections called partitions, such that a program executing on a thread in one partition cannot interfere with any program executing on a thread in a different partition. This isolation can be provided for both problem state and privileged state programs, by using a layer of trusted software, called a hypervisor program (or simply a “hypervisor”), and the resources provided by this facility to manage system resources. (A hypervisor is a program that runs in hypervisor state; see below.)

The number of partitions supported is implementation-dependent.

A thread is assigned to one partition at any given time. A thread can be assigned to any given partition without consideration of the physical configuration of the system (e.g., shared registers, caches, organization of the storage hierarchy), except that threads that share certain hypervisor resources may need to be assigned to the same partition; see Section 2.7. The registers and facilities used to control Logical Partitioning are listed below and described in the following subsections.

Except in the following subsections, references to the “operating system” in this document include the hypervisor unless otherwise stated or obvious from context.

2.2 Logical Partitioning Control Register (LPCR)

The layout of the Logical Partitioning Control Register (LPCR) is shown in Figure 1 below.

![Logical Partitioning Control Register](image)

The contents of the LPCR control a number of aspects of the operation of the thread with respect to a logical partition. Below are shown the bit definitions for the LPCR.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:3</td>
<td>Virtualization Control (VC)</td>
</tr>
<tr>
<td>0:1</td>
<td>Virtualized Partition Memory (VPM)</td>
</tr>
</tbody>
</table>

This field controls whether VPM mode is enabled as specified below. (See Section 5.7.3.4 and Section 5.7.2, “Virtualized Partition Memory (VPM) Mode” for additional information on VPM mode.)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This bit controls whether VPM mode is enabled when address translation is disabled</td>
</tr>
<tr>
<td></td>
<td>0 - VPM mode disabled</td>
</tr>
<tr>
<td></td>
<td>1 - VPM mode enabled</td>
</tr>
</tbody>
</table>
1 This bit controls whether VPM mode is enabled when address translation is enabled.
   0 - VPM mode disabled
   1 - VPM mode enabled

2 **Ignore SLB Large Page Specification (ISL)**
   Controls whether ISL mode is enabled as specified below.
   0 - ISL mode disabled
   1 - ISL mode enabled
   When ISL mode is enabled and address translation is enabled, address translation is performed as if the contents of SLB_LILP were 0b000. When address translation is disabled, the setting of the ISL bit has no effect. ISL mode has no effect on SLB, TLB, and ERAT entry invalidations caused by slbie, slbia, tlbia, tlbie, and slbie.

3 **Key-Based Virtualization (KBV)**
   Controls whether Key-Based Virtualization is enabled as specified below.
   0 - KBV is disabled
   1 - KBV is enabled
   When KBV is enabled, Virtual Page Class Key Storage Protection exceptions that occur on operand accesses when VPM1=0 cause Hypervisor Data Storage interrupts.

**Programming Note**

Key-Based Virtualization provides an efficient means for the hypervisor to intercept storage references, e.g., MMIO, that must be emulated. (The corresponding behavior for instruction fetching is not desired.) Virtual Page Class Key Storage Protection exceptions not handled by the hypervisor should be reflected to the operating system at its Data Storage interrupt vector with the hypervisor having set DSISR42.

4:8 Reserved

9:11 **Default Prefetch Depth (DPFD)**
   The DPFD field is used as the default prefetch depth for data stream prefetching when DSCR_PDPF=0; see page 764.

12:16 **Virtual Real Mode Area Segment Descriptor (VRMASD)**
   When address translation is disabled and VPM0=1, the contents of this field specify the L and LP fields of the segment descriptor that apply for storage references to the virtualized real mode area (VRMA). See Section 5.7.3.4 for additional information. The definitions and allowed values of the L and LP fields are the same as for the corresponding fields in the segment descriptor. (See Section 5.7.7.) If VPM0=0 or address translation is enabled, the setting of the VRMASD has no effect.

**Bit Description**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Virtual Page Size Selector Bit 0 (L)</td>
</tr>
<tr>
<td>1:2</td>
<td>Reserved</td>
</tr>
<tr>
<td>3:4</td>
<td>Virtual Page Size Selector Bits 1:2 (LP)</td>
</tr>
</tbody>
</table>

**Programming Note**

Specifying that LILP=0b000 in the VRMASD field when VPM mode is enabled has the same effect on address translation when translation is disabled as enabling ISL mode when translation is enabled.

ISL mode is needed when translation is enabled because translation uses the SLB, and the contents of the SLB are accessible to the operating system and should not be modified by the hypervisor. ISL mode is not needed when translation is disabled since translation uses the VRMASD, which is not visible to the operating system and is in complete control of the hypervisor.

17:33 Reserved

34:37 **Real Mode Limit Selector (RMLS)**
   The RMLS field specifies the largest effective address that can be used by partition software when address translation is disabled. The valid RMLS values are implementation-dependent, and each value corresponds to a maximum effective address of $2^m$, where $m$ has a minimum value of 12 and a maximum value equal to the number of bits in the real address size supported by the implementation.

38 **Interrupt Little-Endian (ILE)**
   The contents of the ILE bit are copied into MSR_LE by interrupts that set MSR_HV to 0 (see Section 6.5), to establish the Endian mode for the interrupt handler.

39:40 **Alternate Interrupt Location (AIL)**
   Controls the effective address offset of the interrupt handler and the relocation mode in which it begins execution for all interrupts except Machine Check, System Reset, and Hypervisor Maintenance.
   0 The interrupt is taken with MSR_IR_DR = 0b00 and no effective address offset.
Reserved

2 The interrupt is taken with MSR<sub>IR</sub> = 0b11 and an effective address offset of 0x0000_0000_0001_8000.

3 The interrupt is taken with MSR<sub>IR</sub> = 0b11 and an effective address offset of 0xc000_0000_0000_4000.

Interruptions that cause a transition from MSR<sub>HV</sub>=0 to MSR<sub>HV</sub>=1, or that occur when MSR<sub>IR</sub>=0 or MSR<sub>DR</sub>=0, are always taken as if LPC<sub>AIL</sub>=0.

---

**Programming Note**

One of the purposes of the AIL field is to provide relocation for interrupts that occur while an application is running with MSR<sub>HV</sub> PR=0b11 under a "bare metal" operating system (i.e., an operating system that runs in hypervisor state), such as KVM.

41:44 Reserved

45 **Online** (ONL)

0 The PURR and SPURR do not increment.
1 The PURR and SPURR increment.

---

**Programming Note**

Typically, the hypervisor sets the ONL bit to 0 when the thread is not in a power saving mode, is not performing useful work, and is available for use. The hypervisor may take the state of the ONL bit into account when making course-grain load balancing and power management decisions.

46 Reserved

47:51 **Power-saving mode Exit Cause Enable** (PECE)

47 <S.PC> If PECE<sub>0</sub> = 1 when a Power-Saving Mode instruction is executed, Directed Privileged Doorbell exceptions are enabled to cause exit from power-saving mode; otherwise Directed Privileged Doorbell exceptions are disabled from causing exit from power-saving mode.

48 <S.PC> If PECE<sub>1</sub> = 1 when a Power-Saving Mode instruction is executed, Directed Hypervisor Doorbell exceptions are enabled to cause exit from power-saving mode; otherwise Directed Hypervisor Doorbell exceptions are disabled from causing exit from power-saving mode.

49 If PECE<sub>2</sub> = 1 when a Power-Saving Mode instruction is executed, External exceptions are enabled to cause exit from power-saving mode; otherwise External exceptions are disabled from causing exit from power-saving mode.

50 If PECE<sub>3</sub> = 1 when a Power-Saving Mode instruction is executed, Decrementer exceptions are enabled to cause exit from power-saving mode; otherwise Decrementer exceptions are disabled from causing exit from power-saving mode. (In sleep and rwwinkle power-saving levels, Decrementer exceptions do not occur if the state of the Decrementer is not maintained and updated as if the thread was not in power-saving mode.)

51 If PECE<sub>4</sub>=1 when a Power-Saving Mode instruction is executed, Machine Check, Hypervisor Maintenance, and certain implementation-specific exceptions are enabled to cause exit from power-saving mode; otherwise Machine Check, Hypervisor Maintenance, and the same implementation-specific exceptions are disabled from causing exit from power-saving mode.

It is implementation-specific whether the exceptions enabled by the PECE field cause exit from sleep and rwwinkle power-saving levels. See Section 6.5.1 and Section 6.5.2 for additional information about exit from power-saving mode.

52 **Mediated External Exception Request** (MER)

0 A Mediated External exception is not requested.
1 A Mediated External exception is requested.

The exception effects of this bit are said to be consistent with the contents of this bit if one of the following statements is true.

- LPC<sub>MER</sub> = 1 and a Mediated External exception exists.
- LPC<sub>MER</sub> = 0 and a Mediated External exception does not exist.

A context synchronizing instruction or event that is executed or occurs when LPC<sub>MER</sub> = 0 ensures that the exception effects of LPC<sub>MER</sub> are consistent with the contents of LPC<sub>MER</sub>. Otherwise, when an instruction changes the contents of LPC<sub>MER</sub>, the exception effects of LPC<sub>MER</sub> become consistent with the new contents of LPC<sub>MER</sub> reasonably soon after the change.

---

The exception effects of this bit are said to be consistent with the contents of this bit if one of the following statements is true.
2.3 Real Mode Offset Register (RMOR)

The layout of the Real Mode Offset Register (RMOR) is shown in Figure 2 below.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:4</td>
<td>RMO</td>
<td>Real Mode Offset</td>
</tr>
</tbody>
</table>

Figure 2. Real Mode Offset Register

All other fields are reserved.

The supported RMO values are the non-negative multiples of $2^s$, where $2^s$ is the smallest implementation-dependent limit value representable by the contents of the Real Mode Limit Selector field of the LPCR.

The contents of the RMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 891 and Section 5.7.4 on page 895.

2.4 Hypervisor Real Mode Offset Register (HRMOR)

The layout of the Hypervisor Real Mode Offset Register (HRMOR) is shown in Figure 3 below.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:4</td>
<td>HRMO</td>
<td>Real Mode Offset</td>
</tr>
</tbody>
</table>

Figure 3. Hypervisor Real Mode Offset Register

All other fields are reserved.

The supported HRMO values are the non-negative multiples of $2^r$, where $r$ is an implementation-dependent value and $12 \leq r \leq 26$.

The contents of the HRMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 891 and Section 5.7.4 on page 895.

See Section 6.5 on page 950 for a description of how the setting of LPES affects the processing of interrupts.
2.5 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 4 below.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:63</td>
<td>LPID</td>
<td>Logical Partition Identifier</td>
</tr>
</tbody>
</table>

Figure 4. Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the thread is assigned, affecting operations necessary to manage the coherency of some translation looksaside buffers (see Section 5.10.1 and Chapter 12.). The number of LPIDR bits supported is implementation-dependent.

Programming Note

On some implementations, software must prevent the execution of a `tlbie` instruction with an LPID operand value which matches the contents of another thread’s LPIDR that is being modified or is the same as the new value being written to the LPIDR. This restriction can be met with less effort if one partition identity is used only on threads on which no `tlbie` instruction is ever executed. This partition can be thought of as the transfer partition used exclusively to move a thread from one partition to another.

2.6 Processor Compatibility Register (PCR) [Category: Processor Compatibility]

The layout of the Processor Compatibility Register (PCR) is shown in Figure 5 below.

![Figure 5. Processor Compatibility Register](image)

A PCR bit may also determine how an instruction field value is interpreted or may define other behavior as specified in the bit definitions below.

- Instructions are treated as illegal instructions,
- SPRs are treated as if they were not defined for the implementation,
- The “reserved SPRs” (see Section 1.3.3 of Book I) are treated as not defined for the implementation,
- Fields in instructions are treated as if they were 0s,
- bits in system registers read back 0s, and `mtspr` operations have no effect on their values.
- `rfebb` instructions have the same effect on bits in system registers that they would if the bits were available.

Architecture Note

When a bit in a system register is made unavailable by the PCR, `mtspr` operations performed on the register in problem state have no effect on the value of the bit regardless of the privilege state in which the register may subsequently be read. When transactional memory is made unavailable by the PCR, however, `rfebb` instructions executed in problem state have the same effect on MSR_T as they would if transactional memory were available. This behavior is specified so that illegal transaction state transitions resulting from changes to BESCRTS made by privileged code will cause TM Bad Thing interrupts when `rfebb` is executed, thereby facilitating program debug.

The PCR has no effect on the setting of the MSR and [H]SR1 by interrupts, and by the [H]rtd and `mtmsr[d]` instructions, except as specified elsewhere in this section.
When facilities that have enable bits in the MSR, FSCR, HFSCR, or MMCR0 are made unavailable by the value in the PCR, they become unavailable in problem state as specified above regardless of whether they are enabled by the corresponding MSR, FSCR, HFSCR, or MMCR0 bit; facility availability interrupts (e.g. [Hypervisor] Facility Available, Vector Unavailable, etc.) do not occur as a result of problem-state accesses even if the corresponding field in the MSR, HFSCR, or MMCR0 makes them unavailable in problem state.

Programming Note

Because the PCR does not prevent mtspr, rfscv, [h]rfid, and mtmsr[d] instructions from setting bits in system registers that the PCR will make unavailable after a transition to problem state, these instructions may cause interrupts in a variety of unexpected ways. For example, consider an operating system that sets SRR1 such that rfid returns to problem state with MSR[TS] nonzero. A TM Bad Thing interrupt will result, despite that TM is made unavailable by the PCR.

Similarly, the PCR does not prevent rfebb instructions from setting bits in system registers that the PCR has made unavailable in problem state, and thus changes to BESCR_T made by privileged code have the potential to subsequently cause illegal transaction state transitions when rfebb is executed in problem state, resulting in the occurrence of TM Bad Thing interrupts.

When facilities that have enable bits in the MSR, FSCR, HFSCR, or MMCR0 are made unavailable by the value in the PCR, they become unavailable in problem state as specified above regardless of whether they are enabled by the corresponding MSR, FSCR, HFSCR, or MMCR0 bit; facility availability interrupts (e.g. [Hypervisor] Facility Available, Vector Unavailable, etc.) do not occur as a result of problem-state accesses even if the corresponding field in the MSR, HFSCR, or MMCR0 makes them unavailable in problem state.

Programming Note

Facilities that can be disabled in problem state by the PCR that also have enable bits in either the MSR or HFSCR include Transactional Memory, the BHRB instructions, event-based branch instructions, TAR, DSCR at SPR 3, SIER, MMCR2, the event-based branch instructions, and certain Floating-Point, Vector, and VSX instructions. When any of these facilities are made unavailable in problem state by the PCR, the corresponding [Hypervisor] Facility Unavailable, Floating-Point Unavailable, Vector, or VSX unavailable interrupts do not occur when the facility is accessed in problem state. Note, however, that the PCR does not affect privileged accesses, and thus any Hypervisor Facility Unavailable, Floating-Point Unavailable, Vector unavailable, or VSX unavailable interrupts that are specified to occur as a result of privileged accesses occur regardless of the PCR value.

The bit definitions for the PCR are shown below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Transactional Memory (TM) [Category: Transactional Memory]</td>
</tr>
</tbody>
</table>
- BESCR, EBBHR, and TAR
- The ability of the or 31,31,31 and or 5,5,5 instructions to change the value of PPRPRI.
- The ability of mtspri instructions that attempt to set PPRPRI to 001 or 101 to change the value of PPRPRI.

0 The listed instructions, facilities, and behaviors are available in problem state.
1 The listed instructions, facilities, and behaviors are unavailable in problem state.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcdadd.</td>
<td>Decimal Add Modulo</td>
<td>VSX</td>
</tr>
<tr>
<td>bcdsub.</td>
<td>Decimal Subtract Modulo</td>
<td>VSX</td>
</tr>
<tr>
<td>fmrgew</td>
<td>Floating Merge Even Word</td>
<td>VSX</td>
</tr>
<tr>
<td>fmrgow</td>
<td>Floating Merge Odd Word</td>
<td>VSX</td>
</tr>
<tr>
<td>lxsiwax</td>
<td>Load VSX Scalar as Integer Word Indexed</td>
<td>VSX</td>
</tr>
<tr>
<td>lxsiwzx</td>
<td>Load VSX Scalar as Integer Word and Zero Indexed</td>
<td>VSX</td>
</tr>
<tr>
<td>lxsspx</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
<td>VSX</td>
</tr>
<tr>
<td>mfsrd</td>
<td>Move From VSR Doubleword</td>
<td>VSX</td>
</tr>
<tr>
<td>mfsvswz</td>
<td>Move From VSR Word and Zero</td>
<td>VSX</td>
</tr>
<tr>
<td>mtvsrd</td>
<td>Move To VSR Doubleword</td>
<td>VSX</td>
</tr>
<tr>
<td>mtvsrwa</td>
<td>Move To VSR Word Algebraic</td>
<td>VSX</td>
</tr>
<tr>
<td>mtvsrzw</td>
<td>Move To VSR Word and Zero</td>
<td>VSX</td>
</tr>
<tr>
<td>stxsiwx</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td>VSX</td>
</tr>
<tr>
<td>stxsspx</td>
<td>Store VSX Scalar Single-Precision Indexed</td>
<td>VSX</td>
</tr>
<tr>
<td>vaddcuq</td>
<td>Vector Add &amp; write Carry Unsigned Quadword</td>
<td>V</td>
</tr>
<tr>
<td>vaddcuq</td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword</td>
<td>V</td>
</tr>
<tr>
<td>vaddcuq</td>
<td>Vector Add Extended Unsigned Quadword Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vadddm</td>
<td>Vector Add Unsigned Doubleword Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vadduqm</td>
<td>Vector Add Unsigned Quadword Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vbpermq</td>
<td>Vector Bit Permute Quadword</td>
<td>V</td>
</tr>
<tr>
<td>vcipher</td>
<td>Vector AES Cipher</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vcipherl</td>
<td>Vector AES Cipher Last</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vclzb</td>
<td>Vector Count Leading Zeros Byte</td>
<td>V</td>
</tr>
<tr>
<td>vclzd</td>
<td>Vector Count Leading Zeros Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vclzh</td>
<td>Vector Count Leading Zeros Halfword</td>
<td>V</td>
</tr>
<tr>
<td>vclzw</td>
<td>Vector Count Leading Zeros Word</td>
<td>V</td>
</tr>
<tr>
<td>vcmpqued</td>
<td>Vector Compare Equal To Unsigned Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vcmpqtd</td>
<td>Vector Compare Greater Than Signed Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vcmpqtd</td>
<td>Vector Compare Greater Than Unsigned Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>veqv</td>
<td>Vector Logical Equivalence</td>
<td>V</td>
</tr>
<tr>
<td>vgbbd</td>
<td>Vector Gather Bits by Bytes by Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vmaxsd</td>
<td>Vector Maximum Signed Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vmaxud</td>
<td>Vector Maximum Unsigned Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vminsd</td>
<td>Vector Minimum Signed Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vminud</td>
<td>Vector Minimum Unsigned Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vmrgew</td>
<td>Vector Merge Even Word</td>
<td>VSX</td>
</tr>
<tr>
<td>vmrgow</td>
<td>Vector Merge Odd Word</td>
<td>VSX</td>
</tr>
<tr>
<td>vmulesw</td>
<td>Vector Multiply Even Signed Word</td>
<td>V</td>
</tr>
<tr>
<td>vmulesw</td>
<td>Vector Multiply Even Unsigned Word</td>
<td>V</td>
</tr>
<tr>
<td>vmulosw</td>
<td>Vector Multiply Odd Signed Word</td>
<td>V</td>
</tr>
<tr>
<td>vmulosw</td>
<td>Vector Multiply Odd Unsigned Word</td>
<td>V</td>
</tr>
<tr>
<td>vmuluwm</td>
<td>Vector Multiply Unsigned Word Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vnand</td>
<td>Vector Logical NAND</td>
<td>V</td>
</tr>
</tbody>
</table>

Table 1: Category: VSX and Vector Instructions Controlled by the v2.06 Bit
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>vncipher</td>
<td>Vector AES Inverse Cipher</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vncipherlast</td>
<td>Vector AES Inverse Cipher Last</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vorc</td>
<td>Vector Logical OR with Complement</td>
<td>V</td>
</tr>
<tr>
<td>vpermxor</td>
<td>Vector Permute and Exclusive-OR</td>
<td>V.RAID</td>
</tr>
<tr>
<td>vpksdss</td>
<td>Vector Pack Signed Doubleword Signed Saturate</td>
<td>V</td>
</tr>
<tr>
<td>vpksdus</td>
<td>Vector Pack Signed Doubleword Unsigned Saturate</td>
<td>V</td>
</tr>
<tr>
<td>vpkudum</td>
<td>Vector Pack Unsigned Doubleword Unsigned Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vpkudus</td>
<td>Vector Pack Unsigned Doubleword Unsigned Saturate</td>
<td>V</td>
</tr>
<tr>
<td>vpmsumb</td>
<td>Vector Polynomial Multiply-Sum Byte</td>
<td>V</td>
</tr>
<tr>
<td>vpmsumd</td>
<td>Vector Polynomial Multiply-Sum Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vpmsumh</td>
<td>Vector Polynomial Multiply-Sum Halfword</td>
<td>V</td>
</tr>
<tr>
<td>vpmsumw</td>
<td>Vector Polynomial Multiply-Sum Word</td>
<td>V</td>
</tr>
<tr>
<td>vpopcntb</td>
<td>Vector Population Count Byte</td>
<td>V</td>
</tr>
<tr>
<td>vpopcntd</td>
<td>Vector Population Count Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vpopcnth</td>
<td>Vector Population Count Halfword</td>
<td>V</td>
</tr>
<tr>
<td>vpopcntw</td>
<td>Vector Population Count Word</td>
<td>V</td>
</tr>
<tr>
<td>vrld</td>
<td>Vector Rotate Left Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vsbox</td>
<td>Vector AES S-Box</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vshasigmad</td>
<td>Vector SHA-512 Sigma Doubleword</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vshasigmaw</td>
<td>Vector SHA-256 Sigma Word</td>
<td>V.Crypto</td>
</tr>
<tr>
<td>vslid</td>
<td>Vector Shift Left Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vsrad</td>
<td>Vector Shift Right Algebraic Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vsr</td>
<td>Vector Shift Right Doubleword</td>
<td>V</td>
</tr>
<tr>
<td>vsbecuq</td>
<td>Vector Subtract &amp; write Carry Unsigned Quadword</td>
<td>V</td>
</tr>
<tr>
<td>vsbecuq</td>
<td>Vector Subtract Extended &amp; write Carry Unsigned Quadword</td>
<td>V</td>
</tr>
<tr>
<td>vsbequm</td>
<td>Vector Subtract Extended Unsigned Quadword Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vsbucdm</td>
<td>Vector Subtract Unsigned Doubleword Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vsbqum</td>
<td>Vector Subtract Unsigned Quadword Modulo</td>
<td>V</td>
</tr>
<tr>
<td>vupkhsw</td>
<td>Vector Unpack High Signed Word</td>
<td>V</td>
</tr>
<tr>
<td>vupkls</td>
<td>Vector Unpack Low Signed Word</td>
<td>V</td>
</tr>
<tr>
<td>xsaddsp</td>
<td>VSX Scalar Add Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsaddsp</td>
<td>VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling</td>
<td>VSX</td>
</tr>
<tr>
<td>xsawdpspn</td>
<td>Scalar Convert Single-Precision to Double-Precision format Non-signalling</td>
<td>VSX</td>
</tr>
<tr>
<td>xsavscvxdsdp</td>
<td>VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsavscvxdsdp</td>
<td>VSX Scalar round and Convert Signed Fixed-Point Doubleword to Single-Precision format</td>
<td>VSX</td>
</tr>
<tr>
<td>xsavscvxdsdp</td>
<td>VSX Scalar Convert Unsinged Fixed-Point Doubleword to Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsavscvxdsdp</td>
<td>VSX Scalar round and Convert Unsigned Fixed-Point Doubleword to Single-Precision format</td>
<td>VSX</td>
</tr>
<tr>
<td>xsdivsp</td>
<td>VSX Scalar Divide Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsaddasp</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsaddasp</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsmsubasp</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsmsubmsp</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xmulsp</td>
<td>VSX Scalar Multiply Single-Precision</td>
<td>VSX</td>
</tr>
</tbody>
</table>

Table 1: Category: VSX and Vector Instructions Controlled by the v2.06 Bit

Chapter 2. Logical Partitioning (LPAR) and Thread Control 851
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsnmaddasp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsnmaddmsp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsnmsubasp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsnmsubmsp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsresp</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsrp</td>
<td>VSX Scalar Round to Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xsrqsrtsp</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xssqtrsp</td>
<td>VSX Scalar Square Root Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xssubsp</td>
<td>VSX Scalar Subtract Single-Precision</td>
<td>VSX</td>
</tr>
<tr>
<td>xxleqv</td>
<td>VSX Logical Equivalence</td>
<td>VSX</td>
</tr>
<tr>
<td>xxlnand</td>
<td>VSX Logical NAND</td>
<td>VSX</td>
</tr>
<tr>
<td>xxlorc</td>
<td>VSX Logical OR with Complement</td>
<td>VSX</td>
</tr>
</tbody>
</table>

Table 1: Category: VSX and Vector Instructions Controlled by the v2.06 Bit
0 The listed instructions, facilities, and behaviors are available in problem state.
1 The listed instructions, facilities, and behaviors are unavailable in problem state.

Version 2.05 (v2.05)

This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.05.
- AMR access using SPR 13
- addg6s
- bperm
- cdtdbc, cbcdtb
- dcffix[.]
- divide[a][.], divideu[a][.], dividwe[a][.], dividweu[a][.]
- isel
- ifiwxz [Category: Floating-Point: Phased-In]
- fctidu[.], fctiduz[.], fctiwnu[.], fctiwuz[.], fcfidu[.], fcfidus[.], ftdiv, fsqrt [Category: Floating-Point: Phased-In]
- ldrbx, stdbrx [Category: 64-bit]
- popcntw, popcntd
- All facilities in Category: VSX

0 The listed instructions, facilities, and behaviors are available in problem state.
1 The listed instructions, facilities, and behaviors are unavailable in problem state.

If this bit is set to 1, then the v2.06 bit must also be set to 1.

Reserved

The initial state of the PCR is all 0s.

Programming Note

Because the PCR has no effect on privileged instructions except as specified above, privileged instructions that are available on newer implementations but not available on older implementations will behave differently when the thread is in problem state. On older implementations, either an Illegal Instruction type Program interrupt or a Hypervisor Emulation Assistance interrupt will occur because the instruction is undefined; on newer implementations, a Privileged Instruction Program interrupt will occur because the instruction is implemented. (On older implementations the interrupt will be an Illegal Instruction type Program interrupt if the implementation complies with a version of the architecture that precedes V. 2.05, or complies with V. 2.05 and does not support the Hypervisor Emulation Assistance category, and will be a Hypervisor Emulation Assistance interrupt otherwise.)

In future versions of the architecture, in general the lowest-order reserved bit of the PCR will be used to control the availability of the instructions and related resources that are new in that version of the architecture; the name of the bit will correspond to the previous version of the architecture (i.e. the newest version in which the instructions and related resources were not available).

In these future versions of the architecture, there will be a requirement that if any bit of the low-order defined bits is set to 1 then all higher-order bits of the defined low-order bits must also be set to 1, and the architecture version with which the implementation appears to comply, in problem state, will be the version corresponding to the name of the lowest-order 1 bit in the set of defined low-order PCR bits, or the current architecture version if none of these bits are 1. Also, in general the highest-order reserved bits will be used to control the availability of sets of instructions and related resources having the requirement that their availability be independent of versions of the architecture.

2.7 Other Hypervisor Resources

In addition to the resources described above, all hypervisor privileged instructions as well as the following resources are hypervisor resources, accessible to software only when the thread is in hypervisor state except as noted below.

- All implementation-specific resources except for privileged non-hypervisor implementation-specific SPRs. (See Section 4.4.4 for the list of the implementation-specific SPRs that are allowed to be privileged non-hypervisor SPRs.) Implementa-
tion-specific registers include registers (e.g., “HID” registers) that control hardware functions or affect the results of instruction execution. Examples include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.

- ME bit of the MSR
- SPRs defined as hypervisor-privileged in Section 4.4.4. (Note: Although the Time Base, the PURR, and the SPURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR and SPURR can be read when the thread is in privileged state.)

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., mtspr) only in hypervisor state (MSRHVPR = 0b10). An attempt to modify the contents of a given hypervisor resource, other than MSRME, in privileged but non-hypervisor state (MSRHVPR = 0b00) causes a Privileged Instruction type Program interrupt. An attempt to modify MSRME in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

### Programming Note

Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSRPR=1) using mtspr causes a Privileged Instruction type Program exception, and similarly for MSRME.

### 2.8 Sharing Hypervisor Resources

Shared SPRs are SPRs that are accessible to multiple threads. Changes to shared SPRs made by one thread are immediately readable (using mfspr) by all other threads sharing the SPR.

The LPIDR and DPDES must appear to software to be shared among threads of a sub-processor (see Section 2.9). If the implementation does not support sub-processors, the LPIDR and DPDES must be shared among all threads of the multi-threaded processor.

Certain other hypervisor resources may be shared among threads. Programs that modify these resources must be aware of this sharing, and must allow for the fact that changes to these resources may affect more than one thread.

The following resources may be shared among threads.

- RMOR (see Section 2.3)
- HRMOR (see Section 2.4)
- LPIDR (see Section 2.5)

- PCR [Category: Processor Control] (see Section 2.6)
- PVR (see Section 4.3.1)
- RPR (see Section 4.3.7)
- SDR1 (see Section 5.7.7.2)
- AMOR (see Section 5.7.9.1)
- HMEER (see Section 6.2.9)
- Time Base (see Section 7.2)
- Virtual Time Base (see Section 7.3)
- Hypervisor Decrementer (see Section 7.5)
- certain implementation-specific registers or implementation-specific fields in architected registers

The set of resources that are shared is implementation-dependent.

Threads that share any of the resources listed above, with the exception of the PVR and the HRMOR, must be in the same partition.

For each field of the LPCR, except the AIL, ONL, HDICE, and MER fields, software must ensure that the contents of the field are identical among all threads that are in the same partition and are in a state such that the contents of the field could have side effects. (E.g., software must ensure that the contents of LPCR_LPES are identical among all threads that are in the same partition and are not in hypervisor state.) For the HDICE field, software must ensure that the contents of the field are identical among all threads that share the Hypervisor Decrementer and are in a state such that the contents of the field could have side effects. There are no identity requirements for the other fields listed in the first sentence of this paragraph.

### 2.9 Sub-Processors

Hardware is allowed to sub-divide a multi-threaded processor into “sub-processors” that appear to privileged programs as multi-threaded processors with fewer threads. Such a multi-threaded processor appears to the hypervisor as a processor with a number of threads equal to the sum of all sub-processor threads, and in which the LPIDR for each sub-processor must appear to be shared among all threads of that sub-processor.

### 2.10 Thread Identification Register (TIR) [Category: Server Processor Control]

The TIR is a 64-bit read-only register that contains the thread number, which is a binary number corresponding to the thread.

For implementations that do not support sub-processors, the thread number of a thread is unique among all thread numbers of threads on the multi-threaded processor.
For implementations that support sub-processors, the value of this register depends on whether it is read in hypervisor or privileged, non-hypervisor state as follows.

- When this register is read in privileged, non-hypervisor state, the thread number is unique among all thread numbers of threads on the sub-processor.
- When this register is read in hypervisor state, the thread number is unique among all thread numbers of threads on the multi-threaded processor.

Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which \( TIR = n \) is referred to as “thread n.”

The layout of the TIR is shown below.

**Figure 6. Thread Identification Register**

Access to the TIR is privileged.

Since the thread number contained in this register is different if it is read in hypervisor from when it is read in privileged, non-hypervisor state in implementations that support sub-processors, the following conventions are used.

- The value returned in privileged, non-hypervisor state is referred to as the “privileged thread number.”
- The value returned in hypervisor state is referred to as the “hypervisor thread number.”

### 2.11 Hypervisor Interrupt Little-Endian (HILE) Bit

The Hypervisor Interrupt Little-Endian (HILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the HILE bit are copied into MSR\(_{LE}\) by interrupts that set MSR\(_{HV}\) to 1 (see Section 6.5), to establish the Endian mode for the interrupt handler. The HILE bit is set, by an implementation-dependent method, during system initialization, and cannot be modified after system initialization.

The contents of the HILE bit must be the same for all threads under the control of a given instance of the hypervisor; otherwise all results are undefined.
Chapter 3. Branch Facility

3.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

3.2 Branch Facility Registers

3.2.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the thread. On interrupt, the MSR bits are altered in accordance with Figure 51 on page 951. The MSR can also be modified by the \texttt{mtmsr[d]}, \texttt{rfid}, and \texttt{hrfid} instructions. It can be read by the \texttt{mfmsr} instruction.


d\hline
MSR\hline
0 & 63
\end{tabular}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{machine_state_register.png}
\caption{Machine State Register}
\end{figure}

Below are shown the bit definitions for the Machine State Register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sixty-Four-Bit Mode (SF)</td>
</tr>
<tr>
<td>0</td>
<td>The thread is in 32-bit mode.</td>
</tr>
<tr>
<td>1</td>
<td>The thread is in 64-bit mode.</td>
</tr>
<tr>
<td>1:2</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Hypervisor State (HV)</td>
</tr>
<tr>
<td>0</td>
<td>The thread is not in hypervisor state.</td>
</tr>
<tr>
<td>1</td>
<td>If MSR\textsubscript{PR}=0 the thread is in hypervisor state; otherwise the thread is not in hypervisor state.</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Split Little Endian (SLE)</td>
</tr>
<tr>
<td>0</td>
<td>Instruction and data storage accesses for the thread are the same and use the value specified by MSR\textsubscript{LE}.</td>
</tr>
<tr>
<td>1</td>
<td>Instruction and data storage accesses for the thread are opposite. Instruction storage accesses use the value specified by MSR\textsubscript{LE}. Data storage accesses use the value specified by \neg MSR\textsubscript{LE}.</td>
</tr>
<tr>
<td>6:28</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Bit & Description \\
\hline
4 & Privilege State (PR) \\
0 & 0 privileged \\
0 & 1 problem \\
1 & 0 hypervisor \\
1 & 1 problem \\
\hline
\end{tabular}
\caption{Bit Definitions for Machine State Register}
\end{table}

Hypervisor state is also a privileged state (MSR\textsubscript{PR} = 0). All references to "privileged state" in the Books include hypervisor state unless otherwise stated or if it is obvious from the context. MSR\textsubscript{HV} can be set to 1 only by the \texttt{Sys-tem Call} instruction and some interrupts. It can be set to 0 only by \texttt{rfid} and \texttt{hrfid}.

It is possible to run an operating system in an environment that lacks a hypervisor, by always having MSR\textsubscript{HV} = 1 and using MSR\textsubscript{HV} \| MSR\textsubscript{PR} = 10 for the operating system (effectively, the OS runs in hypervisor state) and MSR\textsubscript{HV} \| MSR\textsubscript{PR} = 11 for applications.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Bit & Description \\
\hline
4 & Privilege State (PR) \\
0 & 0 privileged \\
0 & 1 problem \\
1 & 0 hypervisor \\
1 & 1 problem \\
\hline
\end{tabular}
\caption{Bit Definitions for Machine State Register}
\end{table}

Programming Note

The privilege state of the thread is determined by MSR\textsubscript{HV} and MSR\textsubscript{PR}, as follows.

<table>
<thead>
<tr>
<th>HV</th>
<th>PR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>privileged</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>problem</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>hypervisor</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>problem</td>
</tr>
</tbody>
</table>

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Bit & Description \\
\hline
4 & Privilege State (PR) \\
0 & 0 privileged |
0 & 1 problem |
1 & 0 hypervisor |
1 & 1 problem |
\hline
\end{tabular}
\caption{Bit Definitions for Machine State Register}
\end{table}

Programming Note

The only instructions that can alter MSR\textsubscript{SLE} are \texttt{mtsl}, \texttt{hrfid}, \texttt{rfid}, and \texttt{mtmsr}.
29:30  **Transaction State** (TS) [Category: Transactional Memory]

00 Non-transactional
01 Suspended
10 Transactional
11 Reserved

Changes to MSR[TS] that are caused by Transactional Memory instructions, and by invocation of the transaction’s failure handler, take effect immediately (even though these instructions and events are not context synchronizing).

31  **Transactional Memory Available** (TM) [Category: Transactional Memory]

0 The thread cannot execute any Transactional Memory instructions or access any Transactional Memory registers.
1 The thread can execute Transactional Memory instructions and access Transactional Memory registers unless the Transactional Memory facility has been made unavailable by some other register.

32:37 Reserved

38  **Vector Available** (VEC) [Category: Vector]

0 The thread cannot execute any vector instructions, including vector loads, stores, and moves.
1 The thread can execute vector instructions unless they have been made unavailable by some other register.

39 Reserved

40  **VSX Available** (VSX)

0 The thread cannot execute any VSX instructions, including VSX loads, stores, and moves.
1 The thread can execute VSX instructions unless they have been made unavailable by some other register.

**Programming Note**

An application binary interface defined to support Category: Vector-Scalar operations should also specify a requirement that MSR.FP and MSR.VEC be set to 1 whenever MSR.VSX is set to 1.

41:47 Reserved

48  **External Interrupt Enable** (EE)

0 External, Decrementer, Performance Monitor< S>, and Privileged Doorbell interrupts <S,PC> are disabled.
1 External, Decrementer, Performance Monitor< S>, and Privileged Doorbell interrupts <S,PC> are enabled.

This bit also affects whether Hypervisor Decrementer, Hypervisor Maintenance, and Directed Hypervisor Doorbell <S,PC> interrupts are enabled; see Section 6.5.12 on page 961, Section 6.5.19 on page 965, and Section 6.5.20 on page 966.

49  **Problem State** (PR)

0 The thread is in privileged state.
1 The thread is in problem state.

**Programming Note**

Any instruction that sets MSR.PR to 1 also sets MSREE, MSRIR, and MSRDR to 1.

50  **Floating-Point Available** (FP) [Category: Floating-Point]

0 The thread cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The thread can execute floating-point instructions unless they have been made unavailable by some other register.

**Machine Check Interrupt Enable** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see Chapter 2., "Logical Partitioning (LPAR) and Thread Control", on page 843.

**Programming Note**

The only instructions that can alter MSR_ME are *rfid* and *hrfid*.

52  **Floating-Point Exception Mode 0** (FE0) [Category: Floating-Point]

See below.

53  **Single-Step Trace Enable** (SE) [Category: Trace]

0 The thread executes instructions normally.
1 The thread generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is a Power-Saving Mode instruction, *hrfid*, *rfid* or a Power-Saving Mode instruction, all of which are never traced. Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state <TM>, is not one of the instructions that is forbidden in Transactional state (e.g., *dcbf*, see Section 5.3.1 of Book II).
54  **Branch Trace Enable (BE)**
[Category: Trace]

0  The thread executes branch instructions normally.
1  The thread generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.

Branch tracing need not be supported on all implementations that support the Trace category. If the function is not implemented, this bit is treated as reserved.

55  **Floating-Point Exception Mode 1 (FE1)**
[Category: Floating-Point]

See below.

56:57  Reserved

58  **Instruction Relocate (IR)**

0  Instruction address translation is disabled.
1  Instruction address translation is enabled.

---

**Programming Note**

See the Programming Note in the definition of MSRPR.

59  **Data Relocate (DR)**

0  Data address translation is disabled. Effective Address Overflow (EAO) (see Book I) does not occur.
1  Data address translation is enabled. EAO causes a Data Storage interrupt.

---

**Programming Note**

See the Programming Note in the definition of MSRPR.

60  Reserved

61  **Performance Monitor Mark (PMM)**
[Category: Server]

---

**Programming Note**

See the Programming Note in the definition of MSRPR.

---

62  **Recoverable Interrupt (RI)**

0  Interrupt is not recoverable.
1  Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 6.4.3, “Interrupt Processing” on page 947, 6.5.1, “System Reset Interrupt” on page 952, and 6.5.2, “Machine Check Interrupt” on page 953.

63  **Little-Endian Mode (LE)**
In the thread are in Big-Endian mode when MSR\_SLE=0. When MSR\_SLE=1, instruction storage accesses are in Big-Endian mode and data storage accesses are in Little-Endian mode.

Instruction and data storage accesses for the thread are in Little-Endian mode when MSR\_SLE=0. When MSR\_SLE=1, instruction storage accesses are in Little-Endian mode and data storage accesses are in Big-Endian mode.

Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise</td>
</tr>
</tbody>
</table>

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

Table 2 shows all the Transaction State transitions that can be requested by rfebb, rfid, hrfid, and mtmsrd. The table covers behavior when TM is enabled by the PCR. For causes of the TM Bad Thing exception when TM is disabled by the PCR, see Section 6.5.9. In the table, the contents of MSR\_TS and MSR\_TM are abbreviated in the form AB, where A represents MSR\_TS (N, T or S) and B represents MSR\_TM (0 or 1). “x” in the “B” position means that the entry covers both MSR\_TM values, with the same value applying in all columns of a given row for a given instance of the transition. (E.g., the first row means that the transition from N0 to N0 is allowed and results in N0, and that the transition from N0 to N1 is allowed and results in N1.) “Input MSR\_TS\_SRTM” in the second column refers to the MSR\_TS and MSR\_TM values supplied by BESCR for rfebb (just the TS value), SRR1 for rfid, HSRR1 for hrfid, or register RS for mtmsrd.

3.2.2 State Transitions Associated with the Transactional Memory Facility [Category: Transactional Memory]

Updates to MSR\_TS and MSR\_TM caused by rfebb, rfid, hrfid, or mtmsrd occur as described in Table 2. The value written, and whether or not the instruction causes an interrupt, are dependent on the current values of MSR\_TS and MSR\_TM, and the values being written to these fields. When the setting of MSR\_TS causes an illegal state transition, a TM Bad Thing type Program interrupt is generated.
### Table 2: Transaction state transitions that can be requested by *rfebb*, *rfid*, *hrfid*, and *mtmsrd*.

<table>
<thead>
<tr>
<th>Current MSR&lt;sub&gt;TS&lt;/sub&gt;MSR&lt;sub&gt;TM&lt;/sub&gt;</th>
<th>Input MSR&lt;sub&gt;TS&lt;/sub&gt;MSR&lt;sub&gt;TM&lt;/sub&gt;</th>
<th>Resulting MSR&lt;sub&gt;TS&lt;/sub&gt; MSR&lt;sub&gt;TM&lt;/sub&gt;</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td>Nx</td>
<td>Nx</td>
<td>May occur in the context of a Transactional Memory type of Facility Unavailable interrupt handler, enabling/disabling transactions for user-level applications.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All others - Illegal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>N0</td>
</tr>
<tr>
<td>T0</td>
<td>N/A</td>
<td></td>
<td>Unreachable state</td>
</tr>
<tr>
<td>S0</td>
<td>N0&lt;sup&gt;2&lt;/sup&gt;</td>
<td>S0</td>
<td>Operating system code that is not TM aware may attempt to set TS and TM to zero, thinking they're reserved bits. Change is suppressed.</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T1</td>
<td>May occur at an <em>rfid</em> returning to an application whose transaction was suspended on interrupt.</td>
</tr>
<tr>
<td></td>
<td>Sx</td>
<td>Sx</td>
<td>This case may occur for an <em>rfid</em> returning to an application whose suspended transaction was interrupted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All others - Illegal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>S0</td>
</tr>
<tr>
<td>N1</td>
<td>Nx</td>
<td>Nx</td>
<td>After a <em>treclaim</em>, the OS dispatches Nx program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All others - Illegal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>N0</td>
</tr>
<tr>
<td>T1</td>
<td>all</td>
<td>N1</td>
<td>Disallowed instructions in Transactional state</td>
</tr>
<tr>
<td>S1</td>
<td>T1</td>
<td>T1</td>
<td>May occur after <em>trechkpt</em>, when returning to an application.</td>
</tr>
<tr>
<td></td>
<td>Sx</td>
<td>Sx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>All others - Illegal&lt;sup&gt;1&lt;/sup&gt;</td>
<td>S0</td>
</tr>
</tbody>
</table>

**Notes:**
1. Generate TM Bad Thing type Program interrupt. “All others” includes all attempts to set MSR<sub>TS</sub> to 0b11 (reserved value).
2. Instruction completes, change to MSR<sub>TM</sub> suppressed, except when attempted by *rfebb*, in which case the result is a TM Bad Thing type Program interrupt.

Table 2: Transaction state transitions that can be requested by *rfebb*, *rfid*, *hrfid*, and *mtmsrd*.
For hrfid, and mtmsrd, the attempted transition from S0 to N0 is suppressed in order that interrupt handlers that are “unaware” of transactional memory, and load an MSR value that has not been updated to take account of transactional memory, will continue to work correctly. (If the interrupt occurs when a transaction is running or suspended, the interrupt will set MSR(TS + TM) to S0. If the interrupt handler attempts to load an MSR value that has not been updated to take account of transactional memory, that MSR value will have TS + TM = N0. It is desirable that the interrupt handler remain in state S0, so that it can return normally to the interrupted transaction.)

The problem solved by suppressing this transition does not apply to rfebb, so for rfebb an attempt to transition from S0 to N0 is not suppressed, and instead causes a TM Bad Thing type Program interrupt.
3.3 Branch Facility Instructions

3.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

<table>
<thead>
<tr>
<th>System Call</th>
<th>SC-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>LEV</td>
</tr>
</tbody>
</table>

The System Call instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

The effective address of the instruction following the System Call instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to zero.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 6.5, “Interrupt Definitions” on page 950. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 1 are reserved. Bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

**Special Registers Altered:**
- SRR0
- SRR1
- MSR

**Programming Note**

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an sc mnemonic with one operand as the basic form, and an sc mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

If LEV=1 the hypervisor is invoked. This is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation should be considered a programming error.
**Return From Interrupt Doubleword**

**XL-form**

```
rfid
```

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>18</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{MSR}_{51} & \leftarrow (\text{MSR}_3 \& \text{SRR1}_{51}) \mid (\neg\text{MSR}_3 \& \text{MSR}_{51}) \\
\text{MSR}_3 & \leftarrow \text{MSR}_3 \& \text{SRR1}_3 \\
\text{if} \ (\text{MSR}_{29:31} \neq 0b010 \ | \ \text{SRR1}_{29:31} \neq 0b000) \ \text{then} \\
\text{MSR}_{29:31} & \leftarrow \text{SRR1}_{29:31} \\
\text{MSR}_{48} & \leftarrow \text{SRR1}_{48} \mid \text{SRR1}_{49} \\
\text{MSR}_{58} & \leftarrow \text{SRR1}_{58} \mid \text{SRR1}_{49} \\
\text{MSR}_{59} & \leftarrow \text{SRR1}_{59} \mid \text{SRR1}_{49} \\
\text{MSR}_{0:2} & 4:28 \ 32 \ 37:41 \ 49:50 \ 52:57 \ 60:63 \leftarrow \text{SRR1}_{0:2} \ 4:28 \ 32 \ 37:41 \ 49:50 \ 52:57 \ 60:63 \\
\text{NIA} & \leftarrow \text{SRR0}_{0:61} \ | \ 0b00 \\
\end{align*}
\]

If MSR\(_3\)=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of SRR1 are not equal to 0b000, then the value of bits 29 through 31 of SRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR\(_{48}\). The result of ORing bits 58 and 49 of SRR1 is placed into MSR\(_{58}\). The result of ORing bits 59 and 49 of SRR1 is placed into MSR\(_{59}\). Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, “Transaction state transitions that can be requested by rfebb, rfid, hrfid, and mtmsrd,” on page 861), or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the rfid instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0\(_{61}\) || 0b00 (when SF=1 in the new MSR value) or SRR0\(_{32:61}\) || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

**Special Registers Altered:**

\[
\begin{align*}
\text{MSR} \\
\end{align*}
\]

---

**Programming Note**

If this instruction sets MSR\(_{PR}\) to 1, it also sets MSR\(_{EE}\), MSR\(_{IR}\), and MSR\(_{DR}\) to 1.
Hypervisor Return From Interrupt
Doubleword XL-form

hrfid

\[
\text{Table 3.1: Register Usage for HR RFID}
\]

<table>
<thead>
<tr>
<th>MSR</th>
<th>HSRR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>274</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{Table 2: Transaction State Transitions that can be requested by \textbf{rfbb}, \textbf{rfid}, \textbf{hrfid}, and \textbf{mtmsrd.}}
\]

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of HSRR1 are not equal to 0b000, then the value of bits 29 through 31 of HSRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of HSRR1 is placed into MSR48. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR58. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR59. Bits 0:28, 32, 37:41, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, “Transaction state transitions that can be requested by rfebb, rfid, hrfid, and mtmsrd,” on page 861), or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the \textbf{hrfid} instruction.

Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address HSRR0:061 || 0b00 (when SF=1 in the new MSR value) or 320 || HSRR0:32:61 || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

- MSR

\[
\text{Programming Note}
\]

If this instruction sets MSR\text{PR} to 1, it also sets MSR\text{EE}, MSR\text{IR}, and MSR\text{DR} to 1.
3.3.2 Power-Saving Mode Instructions

The Power-Saving Mode instructions provide a means by which the hypervisor can put the thread into power-saving mode. When the thread is in power-saving mode it does not execute instructions, and it may consume less power than it would consume when it is not in power-saving mode.

There are four levels of power-savings, called doze, nap, sleep, and rvwinkle. For each level in this list, the power consumed is less than or equal to the power consumed in the preceding level, and the time required for the thread to exit from the level and for software then to resume normal operation is greater than or equal to the corresponding time for the preceding level. Doze power-saving level requires a minimum amount of such time, while the other levels may require more time. Resources other than those listed in the instruction descriptions that are maintained in each level other than doze, and the actions required by the hypervisor in order for software to resume normal operation after the thread exits from power-saving mode, are implementation-specific.

Read-only resources (including the HILE bit) are maintained in all power-saving levels. Descriptions of resource state loss in the Power-Saving Mode instruction descriptions do not apply to read-only resources.

---

**Programming Note**

The hypervisor determines which power-saving level to enter based on how responsive the system needs to be. If the hypervisor decides that some loss of state is acceptable, it can use the nap instruction rather than the doze instruction, and when the thread exits from power-saving mode the hypervisor can quickly determine whether any resources need to be restored.
The thread is placed into doze power-saving level.

When the thread is in doze power-saving level, the state of all thread resources is maintained as if the thread was not in power-saving mode.

When the interrupt that causes exit from doze power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:
None

Programming Note
If the state of the Decrementer were not maintained and updated as if the thread was not in power-saving mode, Decrementer exceptions would not reliably cause exit from nap power-saving level even if Decrementer exceptions were enabled to cause exit.
### Sleep

**XL-form**

```
sleep

19  ///  ///  ///  466  /
0  6  11  16  21  31
```

The thread is placed into sleep power-saving level.

When the thread is in sleep power-saving level, the state of all resources may be lost except for the HRMOR.

When the interrupt that causes exit from sleep power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

**Special Registers Altered:**

None

**Programming Note**

If the state of the Decrementer is not maintained and updated, in sleep or rvwinkle power-saving level, as if the thread was not in power-saving mode, Decrementer exceptions will not reliably cause exit from power-saving mode even if Decrementer exceptions are enabled to cause exit.

**Note**

See the Notes that appear in the *rvwinkle* instruction description.

### Rip Van Winkle

**XL-form**

```
rwinkle

19  ///  ///  ///  498  /
0  6  11  16  21  31
```

The thread is placed into rvwinkle power-saving level.

When the thread is in rvwinkle power-saving level, the state of all resources may be lost except for the HRMOR.

When the interrupt that causes exit from rvwinkle power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

**Special Registers Altered:**

None

**Programming Note**

In the short story by Washington Irving, Rip Van Winkle is a man who fell asleep on a green knoll and awoke twenty years later.

**Note**

See the Notes that appear in the *sleep* instruction description.
3.3.2.1 Entering and Exiting Power-Saving Mode

In order to enter power-saving mode, the hypervisor must use the instruction sequence shown below. Before executing this sequence, the hypervisor must ensure that LPCR\textsubscript{MER} contains the value 0, the LPCR\textsubscript{PECE} contains the desired value if doze or nap power-saving level is to be entered, MSR\textsubscript{SF}, MSR\textsubscript{HV}, and MSR\textsubscript{ME} contain the value 1, and all other bits of the MSR contain the value 0 except for MSR\textsubscript{RI}, which may contain either 0 or 1. Depending on the implementation and on the power-saving mode being entered, it may also be necessary for the hypervisor to save the state of certain resources before entering the sequence. The sequence must be exactly as shown, with no intervening instructions, except that any GPR may be used as Rx and as Ry, and any value may be used for “save_area” provided the resulting effective address is double-word aligned and corresponds to a valid real address.

```
std    Rx, save_area(Ry)    /* last store necessary to save state*/
ptesync  /* order load after*/
         /* last store */
ld     Rx, save_area(Ry)    /* reload from last location, */
         /* for synchronization */
loop:
cmp    Rx, Rx              /* create dependency */
bne   loop
nap/doze/sleep/rvwinkle  /* enter power-saving mode */
b $                   /* branch to self */
```

After the thread has entered power-saving mode as specified above, various exceptions may cause exit from power-saving mode. The exceptions include, System Reset, Machine Check, Decrementer, External, Hypervisor Maintenance, and implementation-specific exceptions. Upon exit from power-saving mode, if the exception was a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the type of exception that caused exit from power-saving mode. See Section 6.5.1 for additional information.

**Programming Note**

The `ptesync` instruction (see Book III-S, Section 5.9.2) in the preceding sequence, in conjunction with the `ld` instruction and the loop, ensure that all storage accesses associated with instructions preceding the `ptesync` instruction, and all Reference, and Change bit updates associated with additional address translations that were performed, by the thread executing the `ptesync` instruction, before the `ptesync` instruction is executed, have been performed with respect to all threads and mechanisms, to the extent required by the associated Memory Coherence Required attributes, before the thread enters power-saving mode. The `b` instruction (branch to self) is not executed since the preceding Power-Saving Mode instruction puts the thread in a power-saving mode in which instructions are not executed. Even though it is not executed, requiring it to be present simplifies implementation and testing because it reduces the synchronization needed between execution of the instruction stream and entry into power-saving mode.

If the performance monitor is in use when the thread enters power-saving mode, the Performance Monitor data obtainable when the thread exits from power-saving mode may be incomplete or otherwise misleading.

**Programming Note**

Software is not required to set the RI bit to any particular value prior to entering power-saving mode because the setting of SRR1\textsubscript{62} upon exit from power-saving mode is independent of the value of the RI bit upon entry into power-saving mode.
3.4 Event-Based Branch Facility and Instruction

The Event-Based Branch facility is described in Chapter 7 of Book II, but only at the level required by the application program.

Event-based branches can only occur in problem state and when event-based branches has been enabled in the FSCR and HFSCR. If MSRPR=0, BESCRQE=1, and an event-based exception exists, the corresponding event-based branch does not occur until MSRPR=1, FSCR_EBB=1, and HFSCR_EBB=1.

If the rfebb instruction attempts to cause a transition to Transactional or Suspended state when PCRTM=1 or an illegal transaction state transition (see Section 3.2.2), a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism is the address of the rfebb instruction.)
Chapter 4. Fixed-Point Facility

4.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

4.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the `mfspr` (page 885) and `mtspr` (page 884) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

4.3 Fixed-Point Facility Registers

4.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the implementation. The contents of the PVR can be copied to a GPR by the `mfspr` instruction. Read access to the PVR is privileged; write access is not provided.

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

**Version**
A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations, such as which categories are supported.

**Revision**
A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA process. Revision numbers are assigned by an implementation-defined process.

4.3.2 Chip Information Register

The Chip Information Register (CIR) is a 32-bit read-only register that contains a value identifying the manufacturer and other characteristics of the chip on which the processor is implemented. The contents of the CIR can be copied to a GPR by the `mfspr` instruction. Read access to the CIR is privileged; write access is not provided.

**Bit Description**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:35</td>
<td>Manufacturer ID (ID) A four-bit field that identifies the manufacturer of the chip.</td>
</tr>
<tr>
<td>36:63</td>
<td>Implementation-dependent.</td>
</tr>
</tbody>
</table>

Figure 9. Chip Information Register

4.3.3 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be copied to a GPR by the...
mfspr instruction. Read access to the PIR is privileged; write access is not provided.

Figure 10. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.

4.3.4 Control Register

The Control Register (CTRL) is a 32-bit register as shown below.

<table>
<thead>
<tr>
<th>32</th>
<th>48</th>
<th>56</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>///</td>
<td>TS</td>
<td>///</td>
<td>RUN</td>
</tr>
</tbody>
</table>

Figure 11. Control Register

The field definitions for the CTRL are shown below.

### Bit(s) Description

| 32:47 | Reserved |
| 48:55 | Thread State (TS) |
| 56:62 | Reserved |
| 63 | RUN |

**Description**

- **Thread State (TS)**
  - Problem State Access
  - Privileged accesses
    - Bits 0:7 of this field are read-only bits that indicate the state of CTRL_RUN for threads with privileged thread numbers 0 through 7, respectively; bits corresponding to privileged thread numbers higher than the maximum privileged thread number supported are set to 0s.
  - Hypervisor accesses
    - Bits 0:7 of this field are read-only bits that indicate the state of CTRL_RUN for threads with hypervisor thread numbers 0 through 7, respectively; bits corresponding to hypervisor thread numbers higher than the maximum hypervisor thread number supported are set to 0s.

- **RUN**
  - This bit controls an external I/O pin. This signal may be used for the following:
    - Driving the RUN Light on a system operator panel
    - Direct External exception routing
    - Performance Monitor Counter incrementing (See Chapter 9.)

The RUN bit can be used by the operating system to indicate when the thread is doing useful work.

Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

4.3.5 Program Priority Register

Privileged programs may set a wider range of program priorities in the PRI field of PPR and PPR32 than may be set by problem-state programs (see Chapter 3 of Book II). Problem-state programs may only set values in the range of 0b001 to 0b100 unless the Problem
State Priority Boost register (see Section 4.3.6) allows the value 0b101. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. For all priorities except 0b101, if a program attempts to set a value that is not allowed for its privilege level, the PRI field remains unchanged. If a problem-state program attempts to set its priority value to 0b101 when this priority value is not allowed for problem state programs, the priority is set to 0b100. The values and their corresponding meanings are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:13</td>
<td>Program Priority (PRI)</td>
<td>001 very low, 010 low, 011 medium low, 100 medium, 101 medium high, 110 high, 111 very high</td>
</tr>
</tbody>
</table>

### 4.3.6 Problem State Priority Boost Register

The Problem State Priority Boost (PSPB) register is a 32-bit register that controls whether problem-state programs have access to program priority medium high. (See Section 3.1 of Book II.)

Figure 12. Problem-State Priority Boost Register

A problem state program is able to set the program priority to medium high only when the PSPB of the thread contains a non-zero value. The maximum value to which the PSPB can be set must be a power of 2 minus 1. Bits that are not required to represent this maximum value must return 0s when read regardless of what was written to them.

When the PSPB is set to a value less than its maximum value but greater than 0, its contents decrease monotonically at the same rate as the SPURR until its contents minus the amount it is to be decreased are 0 or less when a problem-state program is executing on the thread at a priority of medium high. When the contents of the PSPB minus the amount it is to be decreased are 0 or less, its contents are replaced by 0.

When the PSPB is set to its maximum value or 0, its contents do not change until it is set to a different value.

Whenever the priority of a thread is medium high and either of the following conditions exist, hardware changes the priority to medium:

- the PSPB counts down to 0, or
- PSPB=0 and the privilege state of the thread is changed to problem state (MSR_{PP}=1).

### 4.3.7 Relative Priority Register

The Relative Priority Register (RPR) is a 64-bit register that allows the hypervisor to control the relative priorities corresponding to each valid value of PPR_{PRI}.

<table>
<thead>
<tr>
<th>bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2:7</td>
<td>Relative priority of priority level n: Specifies the relative priority that corresponds to the priority corresponding to PPR_{PRI}=n, where a value of 0 indicates the lowest relative priority and a value of 0b111111 indicates the highest relative priority.</td>
</tr>
</tbody>
</table>

Programming Note

The hypervisor must ensure that the values of the RP_n fields increase monotonically for each n and are of different enough magnitudes to ensure that each priority level provides a meaningful difference in priority.

### 4.3.8 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

<table>
<thead>
<tr>
<th>SPRG0</th>
<th>SPRG1</th>
<th>SPRG2</th>
<th>SPRG3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 14. Software-use SPRs

SPRG0, SPRG1, and SPRG2 are privileged registers. SPRG3 is a privileged register except that the contents may be copied to a GPR in Problem state when accessed using the mispr instruction.
HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs.

Neither the contents of the SPRGs, nor accessing them using `mtspr` or `mfspr`, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by non-hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).

Operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

**Programming Note**

Neither the contents of the HSPRGs, nor accessing them using `mtspr` or `mfspr`, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).
4.4 Fixed-Point Facility Instructions

4.4.1 Fixed-Point Load and Store Caching Inhibited Instructions

The storage accesses caused by the instructions described in this section are performed as though the specified storage location is Caching Inhibited and Guarded. The instructions can be executed only in hypervisor state. Software must ensure that the specified storage location is not in the caches. If the specified storage location is in a cache, the results are undefined.

The Fixed-Point Load and Store Caching Inhibited instructions must be executed only when MSR_D=0. The storage location specified by the instructions must not be in storage specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If either of these conditions is violated, the result is a data storage interrupt.

---

**Programming Note**

The instructions described in this section can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches.

See also, in Book I, the introductions to Section 3.3.1, “Fixed-Point Storage Access Instructions”, Section 3.3.2, “Fixed-Point Load Instructions”, and Section 3.3.3, “Fixed-Point Store Instructions”. 
Load Byte and Zero Caching Inhibited Indexed X-form

\textbf{lbzcix RT,RA,RB}

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>853</th>
<th>/</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 560 \mid \mid \text{MEM}(EA, 1) \)

Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into RT56:63. RT0:55 are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None

Load Halfword and Zero Caching Inhibited Indexed X-form

\textbf{lhzcix RT,RA,RB}

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>821</th>
<th>/</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 480 \mid \mid \text{MEM}(EA, 2) \)

Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into RT48:63. RT0:47 are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None

Load Word and Zero Caching Inhibited Indexed X-form

\textbf{lwzcix RT,RA,RB}

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>789</th>
<th>/</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow 320 \mid \mid \text{MEM}(EA, 4) \)

Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None

Load Doubleword Caching Inhibited Indexed X-form

\textbf{ldcix RT,RA,RB}

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>885</th>
<th>/</th>
</tr>
</thead>
</table>

if RA = 0 then \( b \leftarrow 0 \)
else \( b \leftarrow (RA) \)

\( EA \leftarrow b + (RB) \)
\( RT \leftarrow \text{MEM}(EA, 8) \)

Let the effective address (EA) be the sum (RA|0)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:
None
**Store Byte Caching Inhibited Indexed X-form**

```
stbcix    RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>981</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 1) ← (RS)_{56:63}

Let the effective address (EA) be the sum (RAI0)+ (RB). (RS)_{56:63} are stored into the byte in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None

---

**Store Halfword Caching Inhibited Indexed X-form**

```
sthcix    RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>949</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)_{48:63}

Let the effective address (EA) be the sum (RAI0)+ (RB). (RS)_{48:63} are stored into the halfword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None

---

**Store Word Caching Inhibited Indexed X-form**

```
stwcix    RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>917</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)_{32:63}

Let the effective address (EA) be the sum (RAI0)+ (RB). (RS)_{32:63} are stored into the word in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None

---

**Store Doubleword Caching Inhibited Indexed X-form**

```
stdcix    RS,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)

Let the effective address (EA) be the sum (RAI0)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None
4.4.2 OR Instruction

(or Rx,Rx,Rx) can be used to set PPRPRI (see Section 4.3.5) as shown in Figure 16. For all priorities except medium high, PPRPRI remains unchanged if the privilege state of the thread executing the instruction is lower than the privilege indicated in the figure. For priority medium high, PPRPRI is set to medium if the thread executing the instruction is in problem state and medium high priority is not allowed for problem state programs. (The encodings available to problem-state programs, as well as encodings for additional shared resource hints not shown here, are described in Chapter 3 of Book II.)

Figure 16. Priority levels for or Rx,Rx,Rx

<table>
<thead>
<tr>
<th>Rx</th>
<th>PPRPRI</th>
<th>Priority</th>
<th>Privileged</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>001</td>
<td>very low</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>low</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>011</td>
<td>medium low</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>medium</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>medium high</td>
<td>no/yes(^1)</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>high</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>very high</td>
<td>hypv</td>
</tr>
</tbody>
</table>

\(^1\)This value is privileged unless the Problem State Priority Boost register allows the priority value 0b101 (See Section 4.3.6.)
4.4.3 Transactional Memory Instructions [Category: Transactional Memory]

Privileged software that makes the Transactional Memory Facility available to applications takes on the responsibility of managing the facility's resources and the application's transactional state during interrupt handling, service calls, task switches, and its own use of TM. In addition to the existing instructions like rfid and problem state TM instructions that play a role in this management, **treclaim** and **trechkpt.** may be used, as described below. See Section 3.2.2 for additional information about managing the TM facility and associated state transitions.

**Transaction Reclaim X-form**

treclaim. RA

<table>
<thead>
<tr>
<th>CR0</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>25</th>
<th>22</th>
<th>19</th>
<th>16</th>
<th>13</th>
<th>10</th>
<th>7</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

if MSR$_{TS}$ = 0b10 | MSR$_{PS}$ = 0b01 then
  if RA = 0 then cause <- 0x00000001
  else cause <- GPR(RA)$_{56:63}$ || 0x000001
  if TEXASR$_{PS}$ = 0 then
    Discard transactional footprint
    TMRecordFailure(cause)
    Revert checkpointed registers to pre-transactional values
    Discard all resources related to current transaction
  MSR$_{PS}$ <- 0b00  #Non-transactional

The **treclaim.** instruction frees the transactional facility for use by a new transaction. It sets condition register field 0 to 0 || MSR$_{TS}$ || 0. If the transactional facility is in the Transactional state or Suspended state, failure recording is performed as defined in Section 5.3.2 of Book II. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to GPR(RA)$_{56:63}$ || 0x000001. The checkpointed registers are reverted to their pre-transactional values, and all resources related to the current transaction are discarded, including the transactional footprint (if it wasn’t already discarded for a pending failure).

The transaction state is set to Non-transactional.

If an attempt is made to execute **treclaim.** in Non-transactional state, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.
**Transaction Recheckpoint X-form**

**trechkpt.**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>1006</th>
<th>1</th>
</tr>
</thead>
</table>

CR0 ← 0 || MSR_{TS} || 0

MSR_{TS} ← 0b01

TDOOMED ← 1

checkpoint area ← {checkpointed registers}

The **trechkpt.** instruction copies the current (pre-transacational, saved and restored by the operating system) register state to the checkpoint area. It sets condition register field 0 to 0 || MSR_{TS} || 0. The current values of the checkpointed registers are loaded into the checkpoint area. TDOOMED is set to 0b1.

The transaction state is set to Suspended.

If an attempt is made to execute this instruction in Transactional or Suspended state or when TEXAS-RFS=0, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

**Special Registers Altered**

CR0 TS

### 4.4.4 Move To/From System Register Instructions

The Move To Special Purpose Register and Move From Special Purpose Register instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in privileged state. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given below, the “defined” SPR numbers are the SPR numbers shown in the figure for the instruction and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers.

SPR numbers that are not shown in Figure 17 and are in the ranges shown below are reserved for implementation-specific uses.

- 848 - 863
- 880 - 895
- 976 - 991
- 1008 - 1023

Implementation-specific registers must be privileged.

SPR numbers for implementation-specific SPRs should be registered in advance with the Power ISA architects.

**Extended mnemonics**

Extended mnemonics are provided for the **mtspr** and **mfspr** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix A, “Assembler Extended Mnemonics” on page 1015.
### Figure 17. SPR encodings  (Sheet 1 of 3)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR1^</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000 0001</td>
<td>XER</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>00000 00011</td>
<td>DSCR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>00000 0100</td>
<td>LR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>00000 01001</td>
<td>CTR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>13</td>
<td>00000 01101</td>
<td>AMR</td>
<td>no^3</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>17</td>
<td>00000 10001</td>
<td>DSCR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>18</td>
<td>00000 10010</td>
<td>DSISR</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>19</td>
<td>00000 10011</td>
<td>DAR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>22</td>
<td>00000 10110</td>
<td>DEC</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>25</td>
<td>00000 11001</td>
<td>SDR1</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>64</td>
</tr>
<tr>
<td>26</td>
<td>00000 11010</td>
<td>SRR0</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>27</td>
<td>00000 11011</td>
<td>SRR1</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>28</td>
<td>00000 11100</td>
<td>CFAR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>29</td>
<td>00000 11101</td>
<td>AMR</td>
<td>yes^5</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>61</td>
<td>00001 11101</td>
<td>IAMR</td>
<td>yes^5</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>00100 00000</td>
<td>TFHAR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>129</td>
<td>00100 00001</td>
<td>TFIAR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>130</td>
<td>00100 00010</td>
<td>TEXASR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>131</td>
<td>00100 00011</td>
<td>TEXASHU</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>136</td>
<td>00100 01000</td>
<td>CTRL</td>
<td>-</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>152</td>
<td>00100 11000</td>
<td>CTRL</td>
<td>yes</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>153</td>
<td>00100 11100</td>
<td>FSCR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>157</td>
<td>00100 11101</td>
<td>UAMOR</td>
<td>yes^6</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>159</td>
<td>00100 11111</td>
<td>PSPB</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>176</td>
<td>00101 10000</td>
<td>DPDES</td>
<td>hypv^2</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>177</td>
<td>00101 10001</td>
<td>DHDES</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>180</td>
<td>00101 10100</td>
<td>DAWRO</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>186</td>
<td>00101 11010</td>
<td>RPR</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>187</td>
<td>00101 11011</td>
<td>CIABR</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>188</td>
<td>00101 11100</td>
<td>DAWRX0</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>32</td>
</tr>
<tr>
<td>190</td>
<td>00101 11110</td>
<td>HFSCR</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>01000 00000</td>
<td>VRSAVE</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>259</td>
<td>01000 00011</td>
<td>SPRG3</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>268</td>
<td>01000 01100</td>
<td>TB</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>269</td>
<td>01000 01101</td>
<td>TBU</td>
<td>-</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>272-275</td>
<td>01000 100xx</td>
<td>SPRG[0-3]</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>282</td>
<td>01000 11100</td>
<td>EAR</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>32</td>
</tr>
<tr>
<td>283</td>
<td>01000 11101</td>
<td>CIR</td>
<td>-</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>284</td>
<td>01000 11100</td>
<td>TBL</td>
<td>hypv^2</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>285</td>
<td>01000 11101</td>
<td>TBU</td>
<td>hypv^2</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>286</td>
<td>01000 11110</td>
<td>TBU40</td>
<td>hypv</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>287</td>
<td>01000 11111</td>
<td>PVR</td>
<td>-</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>304</td>
<td>01001 10000</td>
<td>HSPPR0</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>305</td>
<td>01001 10001</td>
<td>HSPPR1</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>306</td>
<td>01001 10010</td>
<td>HDSPR0</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>32</td>
</tr>
<tr>
<td>307</td>
<td>01001 10011</td>
<td>HDAR</td>
<td>hypv^2</td>
<td>hypv^2</td>
<td>64</td>
</tr>
<tr>
<td>308</td>
<td>01001 10100</td>
<td>SPURR</td>
<td>hypv^3</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>309</td>
<td>01001 10101</td>
<td>PURR</td>
<td>hypv^3</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>310</td>
<td>01001 10110</td>
<td>HDEC</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>32</td>
</tr>
<tr>
<td>312</td>
<td>01001 11000</td>
<td>RMCR</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>64</td>
</tr>
<tr>
<td>313</td>
<td>01001 11001</td>
<td>HRMCR</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>64</td>
</tr>
<tr>
<td>314</td>
<td>01001 11010</td>
<td>HSRR0</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>64</td>
</tr>
<tr>
<td>315</td>
<td>01001 11011</td>
<td>HSRR1</td>
<td>hypv^3</td>
<td>hypv^3</td>
<td>64</td>
</tr>
</tbody>
</table>
### Figure 17. SPR encodings (Sheet 2 of 3)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR¹</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat²</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>spr₅₉ sp₀₄</td>
<td></td>
<td>mtspr</td>
<td>mfspr</td>
<td></td>
</tr>
<tr>
<td>318</td>
<td>01001 11110</td>
<td>LPCR</td>
<td>hypv³</td>
<td>hypv³</td>
<td>64</td>
</tr>
<tr>
<td>319</td>
<td>01001 11111</td>
<td>LPIDR</td>
<td>hypv³</td>
<td>hypv³</td>
<td>32</td>
</tr>
<tr>
<td>336</td>
<td>01010 10000</td>
<td>HMER</td>
<td>hypv³,⁴</td>
<td>hypv³</td>
<td>64</td>
</tr>
<tr>
<td>337</td>
<td>01010 10001</td>
<td>HMEER</td>
<td>hypv³</td>
<td>hypv³</td>
<td>64</td>
</tr>
<tr>
<td>338</td>
<td>01010 10010</td>
<td>PCR</td>
<td>hypv³</td>
<td>hypv³</td>
<td>64</td>
</tr>
<tr>
<td>339</td>
<td>01010 10011</td>
<td>HEIR</td>
<td>hypv³</td>
<td>hypv³</td>
<td>32</td>
</tr>
<tr>
<td>349</td>
<td>01010 11101</td>
<td>AMOR</td>
<td>hypv³</td>
<td>hypv³</td>
<td>64</td>
</tr>
<tr>
<td>446</td>
<td>01101 11110</td>
<td>TIR</td>
<td>-</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>512</td>
<td>10000 00000</td>
<td>SPEFSCR</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>526</td>
<td>10000 01110</td>
<td>ATB/ATBL</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>527</td>
<td>10000 01111</td>
<td>ATBU</td>
<td>-</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>768</td>
<td>11000 00000</td>
<td>SIER</td>
<td>-</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>769</td>
<td>11000 00001</td>
<td>MMCR2</td>
<td>no⁴</td>
<td>no⁴</td>
<td>64</td>
</tr>
<tr>
<td>770</td>
<td>11000 00010</td>
<td>MMCR4</td>
<td>no⁴</td>
<td>no⁴</td>
<td>64</td>
</tr>
<tr>
<td>771</td>
<td>11000 00011</td>
<td>PMCT</td>
<td>no⁴</td>
<td>no⁴</td>
<td>32</td>
</tr>
<tr>
<td>772</td>
<td>11000 00100</td>
<td>PMC2</td>
<td>no⁴</td>
<td>no⁴</td>
<td>32</td>
</tr>
<tr>
<td>773</td>
<td>11000 00101</td>
<td>PMC3</td>
<td>no⁴</td>
<td>no⁴</td>
<td>32</td>
</tr>
<tr>
<td>774</td>
<td>11000 00110</td>
<td>PMC4</td>
<td>no⁴</td>
<td>no⁴</td>
<td>32</td>
</tr>
<tr>
<td>775</td>
<td>11000 00111</td>
<td>PMC5</td>
<td>no⁴</td>
<td>no⁴</td>
<td>32</td>
</tr>
<tr>
<td>776</td>
<td>11000 01000</td>
<td>PMC6</td>
<td>no⁴</td>
<td>no⁴</td>
<td>32</td>
</tr>
<tr>
<td>779</td>
<td>11000 01011</td>
<td>MMCR0</td>
<td>no⁴</td>
<td>no⁴</td>
<td>64</td>
</tr>
<tr>
<td>780</td>
<td>11000 01100</td>
<td>SIAR</td>
<td>-</td>
<td>no⁴</td>
<td>64</td>
</tr>
<tr>
<td>781</td>
<td>11000 01101</td>
<td>SDAR</td>
<td>-</td>
<td>no⁴</td>
<td>64</td>
</tr>
<tr>
<td>782</td>
<td>11000 01110</td>
<td>MMCR1</td>
<td>-</td>
<td>no⁴</td>
<td>64</td>
</tr>
<tr>
<td>784</td>
<td>11000 10000</td>
<td>SIER</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>785</td>
<td>11000 10001</td>
<td>MMCR2</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>786</td>
<td>11000 10010</td>
<td>MMCR4</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>787</td>
<td>11000 10011</td>
<td>PMCT</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>788</td>
<td>11000 10100</td>
<td>PMC2</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>789</td>
<td>11000 10101</td>
<td>PMC3</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>790</td>
<td>11000 10110</td>
<td>PMC4</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>791</td>
<td>11000 10111</td>
<td>PMC5</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>792</td>
<td>11000 11000</td>
<td>PMC6</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>795</td>
<td>11000 11011</td>
<td>MMCR0</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>796</td>
<td>11000 11100</td>
<td>SIAR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>797</td>
<td>11000 11101</td>
<td>SDAR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>798</td>
<td>11000 11110</td>
<td>MMCR1</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>800</td>
<td>11001 00000</td>
<td>BESCRS</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>801</td>
<td>11001 00001</td>
<td>BESCRSU</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>802</td>
<td>11001 00010</td>
<td>BESCRRR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>803</td>
<td>11001 00011</td>
<td>BESCRRUL</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>804</td>
<td>11001 00100</td>
<td>EBBHR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>805</td>
<td>11001 00101</td>
<td>EBBRR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>806</td>
<td>11001 00110</td>
<td>BESCR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>808</td>
<td>11001 01000</td>
<td>reserved⁹</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>809</td>
<td>11001 01001</td>
<td>reserved⁹</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>810</td>
<td>11001 01010</td>
<td>reserved⁹</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>811</td>
<td>11001 01011</td>
<td>reserved⁹</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>815</td>
<td>11001 01111</td>
<td>YAR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
</tbody>
</table>

¹ SPR: System Program Register
² Cat: Category
³ Hypervisor
⁴ Non-Hypervisor
⁵ Reserved for future use
⁶ Not used
⁷ Privilege is software-controlled
⁸ Reserved for future use
⁹ Not used
### Figure 17. SPR encodings (Sheet 3 of 3)

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR(^1)</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>848</td>
<td>11010 10000</td>
<td>IC</td>
<td>hypv(^3)  yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>849</td>
<td>11010 10001</td>
<td>VTB</td>
<td>hypv(^3)  yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>896</td>
<td>11100 00000</td>
<td>PPR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>898</td>
<td>11100 00010</td>
<td>PPR32</td>
<td>no</td>
<td>32</td>
<td>B</td>
</tr>
<tr>
<td>1023</td>
<td>11111 11111</td>
<td>PIR</td>
<td>-</td>
<td>32</td>
<td>S</td>
</tr>
</tbody>
</table>

1. This register is not defined for this instruction.
2. Note that the order of the two 5-bit halves of the SPR number is reversed.
3. This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2).
4. This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using `mtspr SPR,RS`.
5. The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the `mtspr` instruction description.
6. The value specified in register RS may be ANDed with the contents of the AMOR before being placed into the UAMOR; see the `mtspr` instruction description.
7. MMCR0PMCC controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 9.4.4 for details.
8. The value specified in Register RS may be masked by the contents of the AMOR before being placed into the IAMR; see the `mtspr` instruction description.
9. Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.

SPR numbers 777-778, 783, 793-794, and 799 are reserved for the Performance Monitor. All other SPR numbers that are not shown above are not implementation-specific are reserved.
Move To Special Purpose Register

**XFX-form**

mtspr SPR,RS

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>spr</th>
<th>467</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

n ← spr5..9 || spr0..4

switch (n)

case(13): if MSRHV PR = 0b10 then
SPR(13) ← (RS)
else
if MSRHV PR = 0b00 then
SPR(13) ← ((RS) & AMOR) | ((SPR(13)) & ¬AMOR)
else
SPR(13) ← (RS) & UAM OR | ((SPR(13)) & ¬UAMOR)

case(29,61): if MSRHV PR = 0b10 then
SPR(n) ← (RS)
else
SPR(n) ← ((RS) & AMOR) | ((SPR(n)) & ¬AMOR)

case (157): if MSRHV PR = 0b10 then
SPR(157) ← (RS)
else
SPR(157) ← (RS) & AMOR

case (336): SPR(336) ← (SPR(336)) & (RS)

case (808, 809, 810, 811):
default: if length(SPR(n)) = 64 then
SPR(n) ← (RS)
else
SPR(n) ← (RS)32:63

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of register RS are placed into the designated Special Purpose Register, except as described in the next four paragraphs. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

When the designated SPR is the Authority Mask Register (AMR), (using SPR 13 or SPR 29), or the designated SPR is the Instruction Authority Mask Register (IAMR), and MSRPR=0b00, the contents of bit positions of register RS corresponding to 1 bits in the Authority Mask Override Register (AMOR) are placed into the corresponding bits of the AMR; the other AMR bits are not modified.

When the designated SPR is the AMR, using SPR 13, and MSRPR=1, the contents of bit positions of register RS corresponding to 1 bits in the User Authority Mask Override Register (UAMOR) are placed into the corresponding bits of the AMR; the other AMR bits are not modified.

When the designated SPR is the UAMOR and MSRPR=0b00, the contents of register RS are ANDed with the contents of the AMOR and the result is placed into the UAMOR.

When the designated SPR is the Hypervisor Maintenance Exception Register (HMER), the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

spr0=1 if and only if writing the register is privileged. Execution of this instruction specifying an SPR number with spr0=1 causes a Privileged Instruction type Program interrupt when MSRPR=1 and, if the SPR is a hypervisor resource (see Figure 17), when MSRHV PR=0b00.

Execution of this instruction specifying an SPR number that is not defined for the implementation, including SPR numbers that are shown in Figure 17 but are in a category that is not supported by the implementation, causes one of the following.

<table>
<thead>
<tr>
<th>if spr0=0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- if MSRPR=1: Hypervisor Emulation Assistance interrupt</td>
</tr>
<tr>
<td>- if MSRPR=0: Hypervisor Emulation Assistance interrupt for SPR 0 and no operation (i.e. the instruction is treated as a no-op) for all other SPRs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>if spr0=1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- if MSRPR=1: Privileged Instruction type Program interrupt</td>
</tr>
<tr>
<td>- if MSRPR=0: no operation (i.e. the instruction is treated as a no-op)</td>
</tr>
</tbody>
</table>

If an attempt is made to execute mtspr specifying a TM SPR in other than Non-transactional state, a TM Bad Thing type Program interrupt is generated.

**Special Registers Altered:**
See Figure 17

**Programming Note**

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 12, “Synchronization Requirements for Context Alterations” on page 1009.
Chapter 4. Fixed-Point Facility

Move From Special Purpose Register

XFX-form

mfspr RT,SPR

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>11 spr</th>
<th>21</th>
<th>339</th>
</tr>
</thead>
</table>

n ← spr_{5:9} | spr_{0:4}

switch (n)

  case(129):
    if (MSRH_{PR} = 0b10)&(TFIARHV_{PR}=MSRH_{PR}) |
    ((MSRH_{PR} = 0b00) & (TFIARHV_{PR} = 0b01)) then
      RT ← SPR(n)
    else
      RT ← 0

  case(808, 809, 810, 811):

  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the designated Special Purpose Register is the TFIAR and TFIAR indicates the failure was recorded in a state more privileged than the current state, register RT is set to zero. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

### Programming Note

Note that when a problem state transaction’s failure is recorded in hypervisor state and there is a subsequent need for a context switch in privileged, non-hypervisor state, an attempt to save TFIAR will result in zeros being saved. This is harmless because if the original application ever tries to read the TFIAR, it would read zeros anyway, since the failure took place in hypervisor state.

- **if spr_0=0:**
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPRs 0, 4, 5, and 6 and no operation (i.e. the instruction is treated as a no-op) for all other SPRs

- **if spr_0=1:**
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e. the instruction is treated as a no-op)

### Special Registers Altered:

None

**Note**

See the Notes that appear with mtspr.

- **spr_0=1** if and only if reading the register is privileged.

  Execution of this instruction specifying an SPR number with spr_0=1 causes a Privileged Instruction type Program interrupt when MSR_{PR}=1 and, if the SPR is a hypervisor resource (see Figure 17), when MSR_{HY_{PR}}=0b00.

  Execution of this instruction specifying an SPR number that is not defined for the implementation, including SPR numbers that are shown in Figure 17 but are in a category that is not supported by the implementation, causes one of the following.

---

Version 2.07
Move To Machine State Register X-form

mtmsr   RS, L

if L = 0 then
    MSR48  ← (RS)48 | (RS)49
    MSR58  ← (RS)58 | (RS)49
    MSR59  ← (RS)59 | (RS)49
else
    MSR48 62  ← (RS)48 62

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR48. The result of ORing bits 58 and 49 of register RS is placed into MSR58. The result of ORing bits 59 and 49 of register RS is placed into MSR59. Bits 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the mtmsr instruction description in this section, references to "mtmsr" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an mtmsr instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If this instruction sets MSRPR to 1, it also sets MSREE, MSRIR, and MSDDR to 1.

This instruction does not alter MSRME or MSREL.
(This instruction does not alter MSRHV because it does not alter any of the high-order 32 bits of the MSR.)

If the only MSR bits to be altered are MSREE, to obtain the best performance L=1 should be used.

Programming Note

If MSREE=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an mtmsrd instruction that sets MSREE to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.8, "Interrupt Priorities" on page 970). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 12.

Programming Note

mtmsr serves as both a basic and an extended mnemonic. The Assembler will recognize an mtmsr mnemonic with two operands as the basic form, and an mtmsr mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

There is no need for an analogous version of the mfmsr instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.
Move To Machine State Register
Doubleword X-form

mtmsrd RS,L

<table>
<thead>
<tr>
<th>31</th>
<th>RS</th>
<th>///</th>
<th>L</th>
<th>///</th>
<th>178</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>32</td>
<td>47</td>
</tr>
</tbody>
</table>

if L = 0 then

if (MSR29:31 ← 0b010 | RS29:31 ← 0b000) then
MSR29:31 ← RS29:31
MSR48 ← (RS)48 | (RS)49
MSR58 ← (RS)58 | (RS)49
MSR59 ← (RS)59 | (RS)49
else
MSR48 62 ← (RS)48 62

The MSR is set based on the contents of register RS and of the L field.

L=0: If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of RS are not equal to 0b000, then the value of bits 29 through 31 of RS is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of register RS is placed into MSR48. The result of ORing bits 58 and 49 of register RS is placed into MSR58. The result of ORing bits 59 and 49 of register RS is placed into MSR59. Bits 0:2, 4:28, 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1: Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, "Transaction state transitions that can be requested by rfebb, rfid, hrfd, and mtmsrd," on page 861), or when TM is disabled by the PCR, a transition to Problem state with an active transaction is generated (unless a higher-priority exception exists (see Section 6.8, "Interrupt Priorities" on page 970). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 12.

mtmsrd serves as both a basic and an extended mnemonic. The Assembler will recognize an mtmsrd mnemonic with two operands as the basic form, and an mtmsrd mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Except in the mtmsrd instruction description in this section, references to ‘mtmsrd’ in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an mtmsrd instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note
If this instruction sets MSRPR to 1, it also sets MSRLE, MSRME, and MSRD to 1.
This instruction does not alter MSREE, MSRIE, or MSRDR.
If the only MSR bits to be altered are MSREE, to obtain the best performance L=1 should be used.

Programming Note
If MSREE=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an mtmsrd instruction that sets MSREE to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.8, "Interrupt Priorities" on page 970). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 12.

Programming Note
mtmsrd serves as both a basic and an extended mnemonic. The Assembler will recognize an mtmsrd mnemonic with two operands as the basic form, and an mtmsrd mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.
Move From Machine State Register

**X-form**

\[
\text{mfmsr} \quad \text{RT}
\]

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>///</th>
<th>///</th>
<th>83</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

\[
\text{RT} \leftarrow \text{MSR}
\]

The contents of the MSR are placed into register RT.

This instruction is privileged.

**Special Registers Altered:**

None
Chapter 5. Storage Control

5.1 Overview

A program references storage using the effective address computed by the hardware when it executes a Load, Store, Branch, or Cache Management instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 5.7.3, in Section 5.7.5 and in the following sections. The real address is what is presented to the storage subsystem.

For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

5.2 Storage Exceptions

A storage exception results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Watchpoint).

In certain cases a storage exception may result in the “restart” of (re-execution of at least part of) a Load or Store instruction. See Section 2.1 of Book II, and Section 6.6 in this Book.

5.3 Instruction Fetch

Instructions are fetched under control of MSR\textsubscript{IR}.

\textbf{MSR\textsubscript{IR}=0}

The effective address of the instruction is interpreted as described in Section 5.7.3.

\textbf{MSR\textsubscript{IR}=1}

The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 5.7.5.

5.3.1 Implicit Branch

Explicitly altering certain MSR bits (using \texttt{mtmsr[\textit{d}]}) or explicitly altering SLB entries, Page Table Entries, or certain System Registers (including the HRMOR, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an \textit{implicit branch}. For example, an \texttt{mtmsrd} instruction that changes the value of MSR\textsubscript{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

5.3.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address $2^{32} - 4$ and is an \texttt{mtmsrd} instruction that changes the contents of MSR\textsubscript{SF}, the effective address of the next sequential instruction is undefined.

\textbf{Programming Note}

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0, or HSRR0, as appropriate to the interrupt, are undefined.

5.4 Data Access

Data accesses are controlled by MSR\textsubscript{DR}.

\textbf{MSR\textsubscript{DR}=0}

The effective address of the data is interpreted as described in Section 5.7.3.
The effective address of the data is translated by the Address Translation mechanism described in Section 5.7.5.

5.5 Performing Operations Out-of-Order

An operation is said to be performed “in-order” if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed “out-of-order” if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as Branch, Trap, System Call, and Return From Interrupt instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, operations are performed out-of-order when resources are available that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, any results of the operation are abandoned (except as described below).

In the remainder of this section, including its subsections, “Load instruction” includes the Cache Management and other instructions that are stated in the instruction descriptions to be “treated as a Load”, and similarly for “Store instruction”.

A data access that is performed out-of-order may correspond to an arbitrary Load or Store instruction (e.g., a Load or Store instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

- Stores
  Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).

- Accessing Guarded Storage
  The restrictions for this case are given in Section 5.8.1.1.
5.7 Storage Addressing

Storage Control Overview
- Real address space size is $2^m$ bytes, $m \leq 60$; see Note 1.
- Real page size is $2^{12}$ bytes (4 KB).
- Effective address space size is $2^{64}$ bytes.
- An effective address is translated to a virtual address via the Segment Lookaside Buffer (SLB).
  - Virtual address space size is $2^n$ bytes, $65 \leq n \leq 78$; see Note 2.
  - Segment size is $2^s$ bytes, $s=28$ or $40$.
  - $2^{n-40} \leq$ number of virtual segments $\leq 2^{n-28}$; see Note 2.
  - Virtual page size is $2^p$ bytes, where $12 \leq p$, and $2^p$ is no larger than either the size of the biggest segment or the real address space; a size of 4 KB, 64 KB, and an implementation-dependent number of other sizes are supported; see Note 3. The Page Table specifies the virtual page size. The SLB specifies the base virtual page size, which is the smallest virtual page size that the segment can contain. The base virtual page size is $2^b$ bytes.
- Segments contain pages of a single size, a mixture of 4 KB and 64 KB pages, or a mixture of page sizes that include implementation-dependent page sizes.
- A virtual address is translated to a real address via the Page Table.

Notes:
1. The value of $m$ is implementation-dependent (subject to the maximum given above). When used to address storage, the high-order 60-$m$ bits of the “60-bit” real address must be zeros.
2. The value of $n$ is implementation-dependent (subject to the range given above). In references to 78-bit virtual addresses elsewhere in this Book, the high-order 78-$n$ bits of the “78-bit” virtual address are assumed to be zeros.
3. The supported values of $p$ for the larger virtual page sizes are implementation-dependent (subject to the limitations given above).

5.7.1 32-Bit Mode
The computation of the 64-bit effective address is independent of whether the thread is in 32-bit mode or 64-bit mode. In 32-bit mode (MSRSF=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses in 32-bit mode differ from those in 64-bit mode.

Programming Note
Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the Power ISA. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the Power ISA. The SLB entries that translate these 16 ESIDs can be used to emulate these Segment Registers.

5.7.2 Virtualized Partition Memory (VPM) Mode
VPM mode enables the hypervisor to reassign all or part of a partition’s memory transparently so that the reassignment is not visible to the partition. When this is done, the partition’s memory is said to be “virtualized.” The VPM field in the LPCR enables VPM mode separately when address translation is enabled and when translation is disabled.

If the thread is not in hypervisor state, and either address translation is enabled and VPM1=1, or address translation is disabled and VPM0=1, conditions that would have caused a Data Storage or an Instruction Storage interrupt if the affected memory were not virtualized instead cause a Hypervisor Data Storage or a Hypervisor Instruction Storage interrupt respectively. Because the Hypervisor Data Storage and Hypervisor Instruction Storage interrupts always put the thread in hypervisor state, they permit the hypervisor to handle the condition if appropriate (e.g., to restore the contents of a page that was reassigned), and to reflect it to the operating system’s Data Storage or Instruction Storage interrupt handler otherwise.

When address translation is enabled, VPM mode has no effect on address translation. When address translation is disabled, addressing is controlled as specified in Section 5.7.3.

5.7.3 Real And Virtual Real Addressing Modes
When a storage access is an instruction fetch performed when instruction address translation is disabled, or if the access is a data access and data address translation is disabled, it is said to be per-
formed in "real addressing mode" if VPM0=0 and the thread is not in hypervisor state. If the thread is in hypervisor state, the access is said to be performed in "hypervisor real addressing mode" regardless of the value of VPM0. If the thread is not in hypervisor state and VPM0=1, the access is said to be performed in "virtual real addressing mode." Storage accesses in real, hypervisor real, and virtual real addressing modes are performed in a manner that depends on the contents of MSR_HV, VPM, VRMASD, HRMOR, RMLS, RMOR (see Chapter 2), bit 0 of the effective address (EA0), and the state of the Real Mode Storage Control Facility as described below. Bits 1:3 of the effective address are ignored.

MSR_HV=1

- If EA0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 5.7.3.1, controls the access.
- If EA0=1, bits 4:63 of the effective address are used as the real address for the access.

MSR_HV=0

- If VPM0=0, the Offset Real Mode Address mechanism, described in Section 5.7.3.2, controls the access.
- If VPM0=1, the Virtual Real Mode Addressing mechanism, described in Section 5.7.3.4, controls the access.

### 5.7.3.1 Hypervisor Offset Real Mode Address

If MSR_HV = 1 and EA0 = 0, the access is controlled by the contents of the Hypervisor Real Mode Offset Register, as follows.

**Hypervisor Real Mode Offset Register (HRMOR)**

Bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the HRMOR, and the 60-bit result is used as the real address for the access. The supported offset values are all values of the form \(i \times 2^r\), where \(0 \leq i < 2^j\), and \(j\) and \(r\) are implementation-dependent values having the properties that \(12 \leq r \leq 26\) (i.e., the minimum offset granularity is 4 KB and the maximum offset granularity is 64 MB) and \(j+r = m\), where the real address size supported by the implementation is \(m\) bits.

---

#### Programming Note

\(\text{EA}_{4:63}\) should equal \(60\times 0\). If this condition is satisfied, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset.

If \(m<60\), \(\text{EA}_{4:63-m}\) and \(\text{HRMOR}_{4:63-m}\) must be zeros.

---

### 5.7.3.2 Offset Real Mode Address

If VPM0=0 and MSR_HV=0, the access is controlled by the contents of the Real Mode Limit Selector and Real Mode Offset Register, as specified below, and the set of storage locations accessible by code is referred to as the Real Mode Area (RMA).

**Real Mode Limit Selector (RMLS)**

If bits 4:63 of the effective address for the access are greater than or equal to the value (limit) represented by the contents of the LPCR\(_{RMLS}\), the access causes a storage exception (see Section 5.7.9.3). In this comparison, if \(m<60\), bits 4:63-m of the effective address may be ignored (i.e., treated as if they were zeros), where the real address size supported by the implementation is \(m\) bits. The supported limit values are of the form \(2^j\), where \(12 \leq j \leq 60\). Subject to the preceding sentence, the number and values of the limits supported are implementation-dependent.

**Real Mode Offset Register (RMOR)**

If the access is permitted by the LPCR\(_{RMLS}\), bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the RMOR, and the low-order \(m\) bits of the 60-bit result are used as the real address for the access. The supported offset values are all values of the form \(i \times 2^s\), where \(0 \leq i < 2^k\), and \(k\) and \(s\) are implementation-dependent values having the properties that \(2^s\) is the minimum limit value supported by the implementation (i.e., the minimum value representable by the contents of the LPCR\(_{RMLS}\)) and \(k+s = m\).

---

#### Programming Note

The offset specified by the RMOR should be a non-zero multiple of the limit specified by the RMLS. If these registers are set thus, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset. (The offset must not be zero, because real page 0 contains the fixed interrupt vectors and real pages 1 and 2 may be used for implementation-specific purposes; see Section 5.7.4, "Address Ranges Having Defined Uses" on page 895.)
5.7.3.3 Storage Control Attributes for Accesses in Real and Hypervisor Real Addressing Modes

Storage accesses in hypervisor real addressing mode are performed as though all of storage had the following storage control attributes, except as modified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.3.1). (The storage control attributes are defined in Book II.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses except those caused by the Load/Store Caching Inhibited instructions; Caching Inhibited, for data accesses caused by the Load/Store Caching Inhibited instructions
- Memory Coherence Required, for data accesses
- Guarded
- not SAO

Storage accesses in real addressing mode are performed as though all of storage had the following storage control attributes. (Such accesses use the Offset Real Mode Address mechanism.)

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required, for data accesses
- not Guarded
- not SAO

Additionally, storage accesses in real or hypervisor real addressing modes are performed as though all storage was not No-execute.

Programming Note

Because storage accesses in real addressing mode and hypervisor real addressing mode do not use the SLB or the Page Table, accesses in these modes bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference and change information is not recorded).

5.7.3.3.1 Hypervisor Real Mode Storage Control

The Hypervisor Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as non-Guarded in hypervisor real addressing mode. For the first technique, the facility provides for the specification, at coarse granularity, of the boundary between non-Guarded and Guarded real storage. Any storage location below the specified boundary is treated as non-Guarded in hypervisor real addressing mode, and any storage location at or above the boundary is treated as Guarded in hypervisor real addressing mode. For the second technique, the facility divides real storage into history blocks, in implementation-specific sizes. The history for instruction fetches is tracked separately from that for data accesses. If there is no instruction fetch history for a block and it is the target of an instruction fetch, the access is performed as though the block is Guarded, but the block is treated as non-Guarded for subsequent instruction fetches on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using a Load/Store Caching Inhibited instruction, the access is performed as though the block is Guarded, and the block is treated as Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using any other Load or Store instruction, the access is performed as though the block is Guarded, but the block is treated as non-Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain.

The storage location specified by a Load/Store Caching Inhibited instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If the second technique is used, the storage location specified by any other Load or Store instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as Guarded. (For the second technique, "specified by the Hypervisor Real Mode Storage Control facility" means "specified in a history block"). For the second technique, the history can be erased using an slbia instruction; see See Section 5.9.3.1.
The facility does not apply to implicit accesses to the Page Table performed during address translation or in recording reference and change information. These accesses are performed as described in Section 5.7.3.5.

The second caution is to avoid storage paradoxes that result from a caching-inhibited access to a location that is held in a cache. The nature of this caution and its solution are described in Section 5.8.2.2, “Altering the Storage Control Bits”. The minimum granularity for concern is the history block, but may be larger, depending on extant translations to the storage in question. Since the consistency of instruction storage is managed by software and hypervisor real mode instruction fetches are always not Caching Inhibited, instruction fetches are also irrelevant to this caution.

The preceding capability can be used to improve the performance of hypervisor software that runs in hypervisor real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded.

5.7.3.4 Virtual Real Mode Addressing Mechanism

If VPM0=1, MSR\textsubscript{e}=0, and MSR\textsubscript{DR}=0 or MSR\textsubscript{IR}=0 as appropriate for the type of access, the access is said to be made in virtual real addressing mode and is controlled by the mechanism specified below. The set of storage locations accessible by code referred to as the Virtualized Real Mode Area (VRMA).

In virtual real addressing mode, address translation, storage protection, and reference and change recording are handled as follows.

- Address translation and storage protection are handled as if address translation were enabled, except that translation of effective addresses to virtual addresses use the SLBE values in Figure 18 instead of the entry in the SLB corresponding to the ESID. In this translation, bits 0:23 of the effective address are ignored (i.e. treated as if they were 0s), bits 24:63-m may be ignored if m < 40, and the Virtual Page Class Key Protection mechanism does not apply.

The Virtual Page Class Key Protection mechanism does not apply because the authority mask that an OS has set for application programs executing with address translation enabled may not be the same as the authority mask required by the OS when address translation is disabled, such as when first entering an interrupt handler.

Reference and change recording are handled as if address translation were enabled.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESID</td>
<td>360</td>
</tr>
<tr>
<td>V</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0b01 - 1 TB</td>
</tr>
<tr>
<td>VSID</td>
<td>0b00</td>
</tr>
<tr>
<td>K\textsubscript{s}</td>
<td>0</td>
</tr>
<tr>
<td>K\textsubscript{p}</td>
<td>undefined</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>VRM\textsubscript{ASD}\textsubscript{L}</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>LP</td>
<td>VRM\textsubscript{ASD}\textsubscript{LP}</td>
</tr>
</tbody>
</table>

Figure 18. SLBE for VRMA
5.7.3.5 Storage Control Attributes for Implicit Storage Accesses

Implicit accesses to the Page Table during address translation and in recording reference and change information are performed as though the storage occupied by the Page Table had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded
- not SAO

The definition of “performed” given in Book II applies also to these implicit accesses; accesses for performing address translation are considered to be loads in this respect, and accesses for recording reference and change information are considered to be stores. These implicit accesses are ordered by the \textit{ptesync} instruction as described in Section 5.9.2.

5.7.4 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- Fixed interrupt vectors
  
  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- Implementation-specific use
  
  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- Offset Real Mode interrupt vectors
  
  The real pages beginning at the real address specified by the HRMOR and RMOR are used similarly to the page for the fixed interrupt vectors.

- Relocated interrupt vectors
  
  Depending on the values of MSR\textsubscript{IR DR} and LPCR\textsubscript{AIL} and on whether the specific interrupt will cause MSR\textsubscript{HV} to change, either the virtual page containing the byte addressed by effective address 0x0000_0000_0001_8000 or the virtual page containing the byte addressed by effective address 0xc000_0000_0000_4000 may be used similarly to the page for the fixed interrupt vectors. (See Section 2.2.)

- Page Table
  
  A contiguous sequence of real pages beginning at the real address specified by SDR1 contains the Page Table.

5.7.5 Address Translation Overview

The effective address (EA) is the address generated by the hardware for an instruction fetch or for a data access. If address translation is enabled, this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage.

The first step in address translation is to convert the effective address to a virtual address (VA), as described in Section 5.7.6. The second step, conversion of the virtual address to a real address (RA), is described in Section 5.7.7.

If the effective address cannot be translated, a storage exception (see Section 5.2) occurs.

Figure 19 gives an overview of the address translation process.
5.7.6 Virtual Address Generation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 20.

5.7.6.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The contents of the SLB are managed by software, using the instructions described in Section 5.9.3.1. See Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009 for the rules that software must follow when updating the SLB.

SLB Entry

Each SLB entry (SLBE, sometimes referred to as a “segment descriptor”) maps one ESID to one VSID. Figure 21 shows the layout of an SLB entry.
All other fields are reserved. B₀ (SLBE₃₇) is treated as a reserved field.

Figure 21. SLB Entry

Instructions cannot be executed from a No-execute (N=1) segment.

Segments may contain a mixture of page sizes. The L and LP bits specify the base virtual page size that the segment may contain. The SLBL||LP encodings are those shown in Figure 22. The base virtual page size (also referred to as the “base page size”) is the smallest virtual page size for the segment. The base virtual page size is 2ᵇ bytes. The actual virtual page size (also referred to as the “actual page size” or “virtual page size”) is specified by PTEL LP.

All other fields are reserved. B₀ (SLBE₃₇) is treated as a reserved field.

Figure 22. Page Size Encodings

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:35</td>
<td>ESID</td>
<td>Effective Segment ID</td>
</tr>
<tr>
<td>36</td>
<td>V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
</tr>
<tr>
<td>37:38</td>
<td>B</td>
<td>Segment Size Selector</td>
</tr>
</tbody>
</table>
| 0b00 - 256 MB (s=28)  
0b01 - 1 TB (s=40)  
0b10 - reserved  
0b11 - reserved |
| 39:88 | VSID | Virtual Segment ID |
| 89    | Kₛ | Supervisor (privileged) state storage key (see Section 5.7.9.2) |
| 90    | Kₚ | Problem state storage key (See Section 5.7.9.2) |
| 91    | N  | No-execute segment if N=1 |
| 92    | L  | Virtual page size selector bit 0 |
| 93    | C  | Class |
| 95:96 | LP | Virtual page size selector bits 1:2 |

For each SLB entry, software must ensure the following requirements are satisfied.

- L||LP contains a value supported by the implementation.
- The base virtual page size selected by the L and LP fields does not exceed the segment size selected by the B field.
- If s=40, the following bits of the SLB entry contain 0s.
  - ESID₂₄:₃₅
  - VSID₈:₉₉

The bits in the above two items are ignored by the hardware.

The Class field of the SLBE is used in conjunction with the slbie and slbia instructions (see Section 5.9.3.1). “Class” refers to a grouping of SLB entries and implementation-specific lookaside information so that only entries in a certain group need be invalidated and others might be preserved. The Class value assigned to an implementation-specific lookaside entry derived from an SLB entry must match the Class value of that SLB entry. The Class value assigned to an implementation-specific lookaside entry that is not derived from an SLB entry (such as real mode address “translations”) is 0.

Software must ensure that the SLB contains at most one entry that translates a given effective address, and that if the SLB contains an entry that translates a given effective address, then any previously existing translation of that effective address has been invalidated. An attempt to create an SLB entry that violates this requirement may cause a Machine Check.

5.7.6.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied for indicated fields in the SLBE.

- V=1
- ESID₀:₆₃=EA₀:₆₃, where the value of s is specified by the B field in the SLBE being tested

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the SLB search succeeds, the virtual address (VA) is formed from the EA and the matching SLB entry fields as follows.

For each SLB entry, software must ensure the following requirements are satisfied.

- L||LP contains a value supported by the implementation.
- The base virtual page size selected by the L and LP fields does not exceed the segment size selected by the B field.
- If s=40, the following bits of the SLB entry contain 0s.
  - ESID₂₄:₃₅
  - VSID₈:₉₉

The bits in the above two items are ignored by the hardware.

The Class field of the SLBE is used in conjunction with the slbie and slbia instructions (see Section 5.9.3.1). “Class” refers to a grouping of SLB entries and implementation-specific lookaside information so that only entries in a certain group need be invalidated and others might be preserved. The Class value assigned to an implementation-specific lookaside entry derived from an SLB entry must match the Class value of that SLB entry. The Class value assigned to an implementation-specific lookaside entry that is not derived from an SLB entry (such as real mode address “translations”) is 0.

Software must ensure that the SLB contains at most one entry that translates a given effective address, and that if the SLB contains an entry that translates a given effective address, then any previously existing translation of that effective address has been invalidated. An attempt to create an SLB entry that violates this requirement may cause a Machine Check.

5.7.6.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied for indicated fields in the SLBE.

- V=1
- ESID₀:₆₃=EA₀:₆₃, where the value of s is specified by the B field in the SLBE being tested

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the SLB search succeeds, the virtual address (VA) is formed from the EA and the matching SLB entry fields as follows.
VA=VSID_{0:77} \parallel EA_{64:63}

The Virtual Page Number (VPN) is bits 0:77-p of the virtual address. The value of p is the actual virtual page size specified by the PTE used to translate the virtual address (see Section 5.7.7.1). If SLBEN = 1, the N (No-execute) value used for the storage access is 1.

If the SLB search fails, a segment fault occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

5.7.7 Virtual to Real Translation

Conversion of a 78-bit virtual address to a real address is done by searching the Page Table as shown in Figure 23.
Figure 23. Translation of 78-bit virtual address to 60-bit real address
5.7.7.1 Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers, where the real page number of a real page is bits 0:47 of the address of the first byte in the real page. The HTAB's size can be any size $2^n$ bytes where $18 \leq n \leq 46$. The HTAB must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 5.7.3.5). The starting address must be a multiple of its size unless the implementation supports the Server.Relaxed Page Table Alignment category, in which case its starting address is a multiple of $2^{18}$ bytes (see Section 5.7.7.4).

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

See Section 5.10 for the rules that software must follow when updating the Page Table.

**Programming Note**

The Page Table must be treated as a hypervisor resource (see Chapter 2), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

### Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 24 shows the layout of a PTE. This layout is independent of the Endian mode of the thread.

<table>
<thead>
<tr>
<th>Dword</th>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
<th>SW</th>
<th>L</th>
<th>H</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0:1</td>
<td>B Segment Size</td>
<td>0b00 - 256 MB</td>
<td>52</td>
<td>55</td>
<td>56</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b01 - 1 TB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b10 - reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0b11 - reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:56</td>
<td></td>
<td>AVA Abbreviated Virtual Address</td>
<td>44</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>57:60</td>
<td></td>
<td>SW Available for software use</td>
<td>52</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>L</td>
<td>Virtual page size</td>
<td>57</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b0 - 4 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1 - greater than 4KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(large page)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>H</td>
<td>Hash function identifier</td>
<td>56</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
<td>57</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note**

The H bit in the Page Table entry should not be set to one unless the secondary Page Table search has been enabled.

If $b<23$, the Abbreviated Virtual Address (AVA) field contains bits 0:54 of the VA. Otherwise bits 0:77-b of the AVA field contain bits 0:77-b of the VA, and bits 78-b:54 of the AVA field must be zero.

**Programming Note**

The AVA field omits the low-order 23 bits of the VA. These bits are not needed in the PTE, because the low-order $b$ of these bits are part of the byte offset into the virtual page and, if $b<23$, the high-order 23-b of these bits are always used in selecting the PTEGs to be searched (see Section 5.7.7.3).

On implementations that support a virtual address size of only $n$ bits, $n<78$, bits 0:77-$n$ of the AVA field must be zeros.

A virtual page is mapped to a sequence of $2^{p-12}$ contiguous real pages such that the low-order $p$-12 bits of the real page number of the first real page in the sequence are 0s.

PTEL LP specify both a base virtual page size (henceforth referred to as the "base page size") and an actual virtual page size (henceforth referred to as the "actual page size" or "virtual page size"). The actual page size is the size of the virtual page mapped by the PTE. The base page size is the smallest actual page size that a segment can contain. See Section 5.7.6.

If PTEL=0, the base virtual page size and actual virtual page size are 4KB, and ARPN concatenated with LP (ARPNNILP) contains the page number of the real page that maps the virtual page described by the entry.

If PTEL=1, the base page size and actual page size are specified by PTELP. In this case, the contents of PTELP have the format shown in Figure 25. Bits labelled "r" are...
bits of the real page number. Bits labelled “z” specify the base page size and actual page size. The values of the “z” bits used to specify each size are implementation-dependent. The values of the “z” bits used to specify each size, along with all possible values of “r” bits in the LP field, must result in LP values distinct from other LP values for other sizes. Actual page sizes 4KB and 64KB are always supported; other actual page sizes are implementation-dependent. If PTEI=1, the actual page size must be greater than 4 KB. Which combinations of different base page size and actual page size are supported is implementation-dependent, except that the combination of a base page size of 4 KB with an actual page size of 64 KB is always supported.

\[
\text{PTE}_{LP} \quad \text{actual page size}
\]

- \(rrrr\_rrrz\) \(\geq 8\) KB
- \(rrrr\_rrzz\) \(\geq 16\) KB
- \(rrrr\_rrzz\) \(\geq 32\) KB
- \(rrrr\_zzzz\) \(\geq 64\) KB
- \(rrrz\_zzzz\) \(\geq 128\) KB
- \(rrzz\_zzzz\) \(\geq 256\) KB
- \(rzzz\_zzzz\) \(\geq 512\) KB
- \(zzzz\_zzzz\) \(\geq 1\) MB

\[
\text{Programming Note}
\]

Instructions cannot be executed from a No-execute (N=1) page.

**Page Table Size**

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs (when the secondary Page Table search is enabled) or more than 8 entries for the same primary PTEG (when the secondary Page Table search is disabled).

While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 5.7.7.2) will reduce the frequency of occurrence of such collisions.

**5.7.7.2 Storage Description**

**Register 1**

The Storage Description Register 1 (SDR1) register is shown in Figure 26.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HTABORG</td>
<td>Real address of Page Table</td>
</tr>
<tr>
<td>46</td>
<td>HTABSIZE</td>
<td>Encoded size of Page Table</td>
</tr>
</tbody>
</table>

All other fields are reserved.

**Figure 26. SDR1**

SDR1 is a hypervisor resource; see Chapter 2.

The HTABORG field in SDR1 contains the high-order 42 bits of the 60-bit real address of the Page Table. The Page Table is thus constrained to lie on a \(2^{18}\) byte (256 KB) boundary at a minimum. At least 11 bits from the hash function (see Figure 23) are used to index into the Page Table. The minimum size Page Table is 256 KB (\(2^{18}\) PTEGs of 128 bytes each).

**Programming Note**

Implementations often have implementation-dependent lookaside buffers (e.g. TLBs and ERATs) used to cache translations of recently used storage addresses. Mapping virtual storage to large pages may increase the effectiveness of such lookaside buffers, improving performance, because it is possible for such buffers to translate a larger range of addresses, reducing the frequency that the Page Table must be searched to translate an address.

**Programming Note**

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is \(2^{31}\) bytes (2 GB), then we have \(2^{31-12}=2^{19}\) real pages. The minimum recommended Page Table size would be \(2^{18}\) PTEGs, or \(2^{25}\) bytes (32 MB).

The actual page size specified by a given PTE\(_{LP}\) format is at least \(2^{12}\times(8-c)\), where \(c\) is the number of \(r\) bits in the format.
The Page Table can be any size $2^n$ bytes where $18 \leq n \leq 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0 unless the implementation supports the Server.Relaxed Page Table Alignment category; see Section 5.7.7.4.

The HTABSIZE field in SDR1 contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form $0b00...011...1$, which is a string of $28 - \text{HTABSIZE}$ 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 23). The number of low-order 0 bits in HTABORG must be greater than or equal to the value in HTABSIZE.

On implementations that support a real address size of only m bits, $m < 60$, bits $0:59-m$ of the HTABORG field are treated as reserved bits, and software must set them to zeros.

### Programming Note

Let $n$ equal the virtual address size (in bits) supported by the implementation. If $n < 67$, software should set the HTABSIZE field to a value that does not exceed $n - 39$. Because the high-order 78-n bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order 67-n bits either all 0s (primary hash; see Section 5.7.7.3) or all 1s (secondary hash). If HTABSIZE > $n-39$, some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

### Example:

Suppose that the Page Table is 16,384 ($2^{14}$) 128-byte PTEGs, for a total size of $2^{21}$ bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3 and the value in HTABORG must have its low-order 3 bits (bits 43:45 of SDR1) equal to 0. This means that the Page Table must begin on a $2^{3+11+7} = 2^{21} = 2$ MB boundary.

### 5.7.7.3 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 5.7.3.5.

An outline of the HTAB search process is shown in Figure 23. If the implementation supports the Server.Relaxed Page Table Alignment category see Section 5.7.7.4. Up to two hash functions are used to locate a PTE that may translate the given virtual address.

1. A 39-bit hash value is computed from the VA. The value of $s$ is the value specified in the SLBE that was used to generate the virtual address; the value of $b$ is equal to $\log_2$(base page size specified in the SLBE that was used to translate the address). **Primary Hash:**

   If $s=28$, the hash value is computed by Exclusive ORing $VA_{11:49}$ with $(11+b0||VA_{50:77-b})$

   If $s=40$, the hash value is computed by Exclusive ORing the following three quantities: $(VA_{24:37}||VA_{38:77-b}), (0||VA_{0:37})$, and $(b-10||VA_{38:77-b})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 4:17 of SDR1 (the high-order 14 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies a particular PTEG, called the “primary PTEG”, whose eight PTEs will be tested.

2. **Secondary Hash:**

   If the secondary Page Table search is enabled (LPCRTE=0), perform the secondary hash function as follows; otherwise do not perform step 2 and proceed to step 3 below.

   If $s=28$, the hash value is computed by taking the ones complement of the Exclusive OR of $VA_{11:49}$ with $(11+b0||VA_{50:77-b})$

   If $s=40$, the hash value is computed by taking the ones complement of the Exclusive OR of the following three quantities: $(VA_{24:37}||VA_{38:77-b}), (0||VA_{0:37})$, and $(b-10||VA_{38:77-b})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 4:17 of SDR1 (the high-order 14 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies the “secondary PTEG”.

3. As many as 8 PTEs in the primary PTEG and, if the secondary Page Table search is enabled, 8 PTEs in the secondary PTEG are tested to determine if any translate the given virtual address. Let $q = \text{minimum}(54, 77-b)$. For a match to exist, the
following conditions must be satisfied, where SLBE is the SLBE used to form the virtual address.

- \( PTE_H = 0 \) for the primary PTEG, \( PTE_H = 1 \) for the secondary PTEG
- \( PTE_V = 1 \)
- \( PTE_B = SLBE_B \)
- \( PTE_AVA[0:q] = VA_0:q \)
- if \( b = 12 \) then
  \( (PTE_L = 0) \land (PTE_LP \text{ specifies the 4KB base page size}) \)
else
  \( (PTE_L = 1) \land (PTE_LP \text{ specifies the base page size specified by } SLBE || LP) \)

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the Page Table search succeeds, the real address (\( RA \)) is formed by concatenating bits 0:59-p of ARPN||LP from the matching PTE with bits 64-p:63 of the effective address (the byte offset), where the p value is the \( \log_2 \) (actual page size specified by \( PTE_L || LP \)).

\[
RA = (ARPN \ || \ LP)[0:59-p] \ || \ EA[64-p:63]
\]

### Programming Note

If \( PTE_L = 0 \), the actual page size (and base page size) are 4 KB. Otherwise the actual page size and base page size are specified by \( PTE_LP \).

Since hardware searches the Page Table using a value of \( b \) equal to \( \log_2 \) (base page size specified in the SLBE that was used to translate the address) regardless of the actual page size, the hardware page table search will identify different PTEs for VAs in different \( 2^b \)-byte blocks of the virtual page if the actual page size is larger than the base page size. Therefore, there may need to be a valid PTE corresponding to each \( 2^b \)-byte block of the virtual page that is referenced. For an actual page size that is larger than \( 2^{23} \) (8 MB), the \( PTE_AVA \) will differ among some or all of these PTEs. Depending on the Page Table size, some or all of these PTEs may be in the same PTEG. Any such PTEs that are in the same PTEG will differ in the value of \( PTE_H \) or \( PTE_AVA \) or both.

All PTEs for the same virtual page should have the same values in the Page Protection, KEY, ARPN, WIMG, and N fields. A set of values from any one of the PTEs that maps the virtual page may be used for an access in the virtual page since lookahead buffer information may be used to translate the virtual address.

To avoid creating multiple matching PTEs, software should not create PTEs for each of two different virtual pages that overlap in the virtual address space. If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where \( 2^p \) is the actual virtual page size of the larger page.

The \( N \) (No-execute) value used for the storage access is the result of ORing the \( N \) bit from the matching PTE with the \( N \) bit from the SLB entry that was used to translate the effective address.

### Programming Note

The value of \( b \) used when searching the Page Table to identify the PTEGs is \( \log_2 \) (base page size). Since a segment may contain pages of different sizes, the hardware searches for PTEs specifying pages of any supported size greater than or equal to the base page size, and the real address is formed using the value of \( p \) specified by the matching PTE.

A virtual page of \( 2^p \) bytes in a segment with a base page size of \( 2^b \) bytes may be mapped by as many as \( 2^{(p-b)} \) PTEs.

If the Page Table search fails, a page fault occurs. This is a Hypervisor Instruction Storage exception or a Hypervisor Data Storage exception, depending on
whether the effective address is for an instruction fetch or for a data access. The N value used for the storage access is the N bit from the SLB entry that was used to translate the effective address.

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. Even though multiple PTEs may be needed for a virtual page whose size is larger than the base page size, one TLB entry derived from a single PTE may be used to translate all of the virtual addresses in the entire virtual page. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 5.10).

In the TLB search, the match criteria include virtual address bits 0:(77-q) where q is an implementation-dependent integer such that \( b \leq q \leq p \). As a result of a Page Table search, multiple matching TLB entries are not created for the same virtual page, except that multiple matching TLB entries may be created if the Page Table contains PTEs that map different-sized virtual pages that overlap in the virtual address space. (If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where \( 2^p \) is the actual virtual page size of the larger page.) If a TLB search finds multiple matching TLB entries created from such PTEs, one of the matching TLB entries is used as if it were the only matching entry, or a Machine Check occurs.

As a result of a Page Table search in a Page Table that does not contain different-sized virtual pages that overlap, it is implementation-dependent whether multiple non-matching TLB entries are created for the same virtual page. However, in this case if multiple TLB entries are created for a given virtual page, at most one matching TLB entry is created for any given virtual address in that virtual page, and q for that TLB entry is less than p.

An implementation may associate each of its TLB entries with the partition for which the TLB entry was created, so that the entries can be retained while other partitions are executing. In this case, when a valid TLB entry is created, the LPID value from LPIDR is written into the TLB entry.

Programming Notes

1. Page Table Entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the `tlbie` or `tlbia` instruction to ensure that the TLB no longer contains a mapping for a particular virtual page.

5.7.4 Relaxed Page Table Alignment [Category: Server.Relaxed Page Table Alignment]

The Page Table can be aligned on any \( 2^{18} \) byte (256 KB) boundary regardless of the HTAB size.

Section 5.7.7.2 describes the Storage Description Register, which includes the HTABORG field. That description generally applies except for the following difference. As the Page Table size is increased beyond 256 KB, the value in HTABORG need not have more of its low-order bits equal to 0. Instead, \( (\text{HTABORG} \parallel 180) \) is the real address of the start of the Page Table regardless of the Page Table size.

A Page Table search is performed as described in Section 5.7.7.3 except the 60-bit real address of a PTEG for both the primary and, if the secondary Page Table search is enabled, the secondary hash is formed by concatenating the following values:

- Bits 0:27 of the 39-bit appropriate primary or secondary hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTAB-SIZE) and then added to the value of bits 4:45 of SDR1 (HTABORG). This part of the real address differs from Section 5.7.7.2.
- Bits 28:38 of the 39-bit hash value.
- Seven 0-bits.

An outline of the PTEG real address computation is shown in Figure 23.

5.7.8 Reference and Change Recording

If address translation is enabled, Reference (R) and Change (C) bits are updated in any one of what could be multiple Page Table Entries that map the virtual page that is being accessed. If the storage operand of a `Load` or `Store` instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent.
accesses to each of the pages for the purpose of setting the Reference and Change bits.

Reference and Change bits are set by the hardware as described below. Setting the bits need not be atomic with respect to performing the access that caused the bits to be updated. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old value for the virtual page, as determined by examining either the Page Table Entry or any lookaside information for the virtual page (e.g., TLB) maintained by the hardware.

Reference Bit

The Reference bit is set to 1 if the corresponding access (load, store, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception, except that the Reference bit is not set to 1 for the access caused by an indexed Move Assist instruction for which the XER specifies a length of zero.

Change Bit

The Change bit is set to 1 if a Store instruction is executed and the store is performed. Otherwise in general the Change bit may be set to 1 if a Store instruction is executed and the store is permitted by the storage protection mechanism and, if the Store instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (see Section 6.4 on page 946)
- Floating-Point Enabled Exception type Program interrupts when the thread is in an Imprecise mode.

The only exception to the preceding statement is that the Change bit is not set to 1 if the instruction is a Store String Indexed instruction for which the XER specifies a length of zero.

Programming Note

A virtual page in a segment with a smaller base page size may be mapped by multiple PTEs. For each access of a virtual page, hardware may search the Page Table to update the R and C bits. If lookaside buffer information for the virtual page already indicates that all such bits to be set have already been set in a PTE that maps the virtual page, hardware need not make an update. Consider the following sequence of events:

1. A virtual page is mapped by 2 PTEs A and B and the R and C bits in both PTEs are 0.
2. A Load instruction accesses the virtual page and the R bit is updated in PTE A.
3. A Load instruction accesses the virtual page and the R bit is updated in PTE B.
4. A Store instruction accesses the virtual page and the C bit is updated in PTE B.
5. The virtual page is paged out. Software must examine both PTE A and B to get the state of the R and C bits for the virtual page.

Furthermore, if in event 2, PTE A was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur. Subsequently, if in event 3 or 4, PTE B was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur.
When the hardware updates the Reference and Change bits in the Page Table Entry, the accesses are performed as described in Section 5.7.3.5, “Storage Control Attributes for Implicit Storage Accesses” on page 895. The accesses may be performed using operations equivalent to a store to a byte, halfword, word, or doubleword, and are not necessarily performed as an atomic read/modify/write of the affected bytes.

These Reference and Change bit updates are not necessarily immediately visible to software. Executing a sync instruction ensures that all Reference and Change bit updates associated with address translations that were performed, by the thread executing the sync instruction, before the sync instruction is executed will be performed with respect to that thread before the sync instruction’s memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multi-threaded systems; see Section 5.10, “Page Table Update Synchronization Requirements” on page 935.

When software refers to a Page Table Entry when MSRDR=1, the Reference and Change bits in the associated Page Table Entry are set as for ordinary loads and stores. See Section 5.10 for the rules software must follow when updating Reference and Change bits.

Figure 27 on page 907 summarizes the rules for setting the Reference and Change bits. The table applies to each atomic storage reference. It should be read from the top down; the first line matching a given situation applies. For example, if stwcx. fails due to both a storage protection violation and the lack of a reservation, the Change bit is not altered.

In the figure, the “Load-type” instructions are the Load instructions described in Books I, II, and III-S, eciwx, and the Cache Management instructions that are treated as Loads. The “Store-type” instructions are the Store instructions described in Books I, II, and III-S, ecowx, and the Cache Management instructions that are treated as Stores. The “ordinary” Load and Store instructions are those described in Books I, II, and III-S. “set” means “set to 1”.

Even though the execution of a Store instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A Store Conditional instruction (stwcx. or stdcx.) is executed, but no store is performed.
- The Store instruction causes a Data Storage exception (for which setting the Change bit is not prohibited).
- The Store instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

For example, when executing a Store instruction, the thread may search the Page Table for the purpose of setting the Change bit and then re-execute the instruction. When reexecuting the instruction, the thread may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another thread, the second search may obtain the new contents, which may preclude the store.

- A system-caused interrupt occurs before the store has been performed.

Because the sync instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the thread executing the sync instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the sync instruction.

If a system-caused interrupt occurs before the store has been performed.

Programming Note

Because the sync instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the thread executing the sync instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the sync instruction.

Even though the execution of a Store instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A Store Conditional instruction (stwcx. or stdcx.) is executed, but no store is performed.
- The Store instruction causes a Data Storage exception (for which setting the Change bit is not prohibited).
- The Store instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

For example, when executing a Store instruction, the thread may search the Page Table for the purpose of setting the Change bit and then re-execute the instruction. When reexecuting the instruction, the thread may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another thread, the second search may obtain the new contents, which may preclude the store.

- A system-caused interrupt occurs before the store has been performed.
### Status of Access

<table>
<thead>
<tr>
<th>Status of Access</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed <em>Move Assist</em> insn w 0 len in XER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Storage protection violation</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Out-of-order I-fetch or Load-type Inst'n (including transactional Load-type inst'n or <em>dcbtst</em>)</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Out-of-order Store-type inst'n, including transactional Store-type inst'n, excluding <em>dcbtst</em></td>
<td>Acc</td>
<td>Acc^1</td>
</tr>
<tr>
<td>Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts^3, or transaction failure</td>
<td>Acc</td>
<td>Acc</td>
</tr>
<tr>
<td>All other cases</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>In-order Load-type or Store-type insn, access not performed^4</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Load-type insn</td>
<td>Acc</td>
<td>No</td>
</tr>
<tr>
<td>Store-type insn</td>
<td>Acc</td>
<td>Acc^2</td>
</tr>
<tr>
<td>Other in-order access</td>
<td>acc</td>
<td>No</td>
</tr>
<tr>
<td>I-fetch</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Ordinary Load, <em>eciw</em></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><em>icbi, icbt, dcbt, dcbtst, dcbst, dcbb[l]</em></td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

^"Acc" means that it is acceptable to set the bit.

1 It is preferable not to set the bit.
2 If C is set, R is also set unless it is already set.
3 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by MSR^FEO FE1.
4 This case does not apply to the *Touch* instructions, because they do not cause a storage access.

---

Figure 27. Setting the Reference and Change bits
5.7.9 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the storage protection mechanism depends on the contents of one or more of the following:

- MSR bits HV, IR, DR, PR
- the key bits in the associated SLB entry
- the page protection bits and key bits in the associated PTE
- the AMR, IAMR, AMOR, and UAMOR
- LPCR bit VPM0

The storage protection mechanism consists of the Virtual Page Class Key Protection mechanism, described in Section 5.7.9.1, and the Basic Storage Protection mechanism, described in Section 5.7.9.2 and Section 5.7.9.3.

When address translation is enabled for an access, the access is permitted if and only if the access is permitted by both the Virtual Page Class Key Protection mechanism and the Basic Storage Protection mechanism. When address translation is disabled for an access, the access is permitted if and only if the access is permitted by the Basic Storage Protection mechanism. If an instruction fetch is not permitted, an Instruction Storage exception or a Hypervisor Instruction Storage exception is generated. If a data access is not permitted, a Data Storage exception or a Hypervisor Data Storage exception is generated.

A protection domain is a maximal range of effective addresses for which variables related to storage protection can be independently specified (including by default, as in real and hypervisor real addressing modes), or a maximal range of addresses, effective or virtual, for which variables related to storage protection cannot be specified. Examples include: a segment, a virtual page (including for a virtualized Real Mode Area), the Real Mode Area (regardless of whether the RMA is virtualized), the effective address range 0:260-1 in hypervisor real addressing mode, and a maximal range of effective or virtual addresses that cannot be mapped to real addresses. A protection boundary is a boundary between protection domains.

5.7.9.1 Virtual Page Class Key Protection

The Virtual Page Class Key Protection mechanism provides the means to assign virtual pages to one of 32 classes, and to modify data access permissions for each class by modifying the Authority Mask Register (AMR), shown in Figure 28, and to modify instruction access permissions for each class by modifying the Instruction Authority Mask Register (IAMR) shown in Figure 29.

**Programming Note**

If address translation is disabled for a given access, the access is not affected by the Virtual Page Class Key Protection mechanism even if the access is made in virtual real addressing mode.

### Authority Mask Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Key0</td>
<td>Access mask for class number 0</td>
</tr>
<tr>
<td>2:3</td>
<td>Key1</td>
<td>Access mask for class number 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2n:2n+1</td>
<td>Keyn</td>
<td>Access mask for class number n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>62:63</td>
<td>Key31</td>
<td>Access mask for class number 31</td>
</tr>
</tbody>
</table>

**Figure 28. Authority Mask Register (AMR)**

The access mask for each class defines the access permissions that apply to loads and stores for which the virtual address is translated using a Page Table Entry that contains a KEY field value equal to the class number. The access permissions associated with each class are defined as follows, where AMR_{2n} and AMR_{2n+1} refer to the first and second bits of the access mask corresponding to class number n.

- A store is permitted if AMR_{2n}=0b0; otherwise the store is not permitted.
- A load is permitted if AMR_{2n+1}=0b0; otherwise the load is not permitted.

The AMR can be accessed using either SPR 13 or SPR 29. Access to the AMR using SPR 29 is privileged.

**Programming Note**

Because the AMR is part of the program context (if address translation is enabled), and because it is desirable for most application programmers not to have to understand the software synchronization requirements for context alterations (or the nuances of address translation and storage protection), operating systems should provide a system library program that application programs can use to modify the AMR.
Instruction Authority Mask Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Resv'd</td>
<td>Access mask for class number 0</td>
</tr>
<tr>
<td>1</td>
<td>Key0</td>
<td>Access mask for class number 1</td>
</tr>
<tr>
<td>2</td>
<td>Resv'd</td>
<td>Access mask for class number 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2n</td>
<td>Resv'd</td>
<td>Access mask for class number n</td>
</tr>
<tr>
<td>2n+1</td>
<td>Keyn</td>
<td>Access mask for class number n+1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>62</td>
<td>Resv'd</td>
<td>Access mask for class number 31</td>
</tr>
<tr>
<td>63</td>
<td>Key31</td>
<td>Access mask for class number 32</td>
</tr>
</tbody>
</table>

Figure 29. Instruction Authority Mask Register (IAMR)

The access mask for each class defines the access permissions that apply to instruction fetches for which the virtual address is translated using a Page Table Entry that contains a KEY field value equal to the class number. The access permission associated with each class is defined as follows, where IAMR_{2n+1} refers to the bit of the access mask corresponding to class number $n$.

- An instruction fetch is permitted if IAMR_{2n+1} = 0b0; otherwise the instruction fetch is not permitted.

Access to the IAMR is privileged.

The Authority Mask Override Register (AMOR) and the User Authority Mask Override Register (UAMOR), shown in Figure 30 and Figure 31 respectively, can be used to restrict modifications (mtspr) of the AMR. Also, the AMOR can be used to restrict modifications of the UAMOR and IAMR. Access to both the AMOR and UAMOR is privileged. The AMOR is a hypervisor resource.

Figure 30. Authority Mask Override Register (AMOR)

Figure 31. User Authority Mask Override Register (UAMOR)

The bits of the AMOR and UAMOR are in 1-1 correspondence with the bits of the AMR (i.e., [U]AMOR$_i$ corresponds to AMR$_i$). The AMOR affects modifications of the AMR and UAMOR in privileged but non-hypervisor state; the UAMOR affects modifications of the AMR in problem state.

Similarly, the odd bits of the AMOR are in 1-1 correspondence with the odd bits of the IAMR (i.e., AMOR$_{2i+1}$ corresponds to IAMR$_{2i+1}$). The AMOR affects modifications of the IAMR in privileged but non-hypervisor state; the IAMR cannot be accessed in problem state.

- When mtspr specifying the AMR (using either SPR 13 or SPR 29) or the IAMR is executed in privileged but non-hypervisor state, the AMOR is used as a mask that controls which bits of the resulting AMR or IAMR contents come from register RS and which AMR or IAMR bits are not modified.
- Similarly, when mtspr specifying the AMR (using SPR 13) is executed in problem state, the UAMOR is used as a mask that controls which bits of the resulting AMR contents come from register RS and which AMR bits are not modified.

A complete description of these effects can be found in the description of the mtspr instruction on page 1053.

Software must ensure that both bits of each even/odd bit pair of the AMOR contain the same value. — i.e., the contents of register RS for mtspr specifying the AMOR must be such that (RS)$_{2n} = (RS)$_{2n+1}$ for every n in the range 0:31 — and likewise for the UAMOR. If this requirement is violated for the UAMOR the results of accessing the UAMOR (including implicitly by the hardware as described in the second item of the preceding list) are boundedly undefined; if the requirement is violated for the AMOR the results of accessing the AMOR (including implicitly by the hardware as described in the first and third items of the list) are undefined.

Version 2.07

Chapter 5. Storage Control 909
The preceding requirement permits designs to implement the AMOR and/or UAMOR as 32-bit registers — specifically, to implement only the even-numbered bits (or only the odd-numbered bits) of the register — in a manner such that the reduction, from the architecturally-required 64 bits to 32 bits, is not visible to (correct) software. This implementation technique saves space in the hardware. (A design that uses this technique does the appropriate “fan in/out” when the register is accessed, to provide the appearance, to (correct) software, of supporting all 64 bits of the register.)

Permitting designs to implement the [U]AMOR as 32-bit registers by virtue of the software requirement specified above, rather than by defining the [U]AMOR as 32-bit registers, permits the architecture to be extended in the future to support controlling modification of the “read access” AMR bits (the odd-numbered bits) independently from the “write access” AMR bits (the even-numbered bits), if that proves desirable. If this independent control does prove desirable, the only architecture change would be to eliminate the software requirement.

When modifying the AMOR and/or UAMOR, the hypervisor should ensure that the two registers are consistent with one another before giving control to a non-hypervisor program. In particular, the hypervisor should ensure that if AMOR$_i$=0 then UAMOR$_i$=0, for all $i$ in the range 0:63. (Having AMOR$_i$=0 and UAMOR$_i$=1 would permit problem state programs, but not the operating system, to modify AMR bit $i$.)
Programming Note

The Virtual Page Class Key Protection mechanism replaces the Data Address Compare mechanism that was defined in versions of the architecture that precede Version 2.04 (e.g., the two facilities use some of the same resources, as described below). However, the Virtual Page Class Key Protection mechanism can be used to emulate the Data Address Compare mechanism. Moreover, programs that use the Data Address Compare mechanism can be modified in a manner such that they will work correctly both on implementations that comply with versions of the architecture that precede Version 2.04 (and hence implement the Data Address Compare mechanism) and on implementations that comply with Version 2.04 of the architecture or with any subsequent version (and hence instead implement the Virtual Page Class Key Protection mechanism). The technique takes advantage of the facts that the SPR number for privileged access to the AMR (29) is the same as the SPR number for the Data Address Compare mechanism's ACCR (Address Compare Control Register), that KEY4 occupies the same bit in the PTE as the Data Address Compare mechanism's AC (Address Compare) bit, and that the definition of ACCR62:63 is very similar to the definition of each even-odd pair of AMR bits. The technique is as follows, where PTE1 refers to doubleword 1 of the PTE.

- Set bits 2:3 and 62:63 of SPR 29 (which is either the ACCR or the AMR) to x, where x is the desired 2-bit value for controlling Data Address Compare matches, and set bits 0:1 to 0s.
- Set PTE154 (which is either the AC bit or KEY4) to the same value that the AC bit would be set to, and set PTE12:3 (which are either RPN bits, that correspond to a real address size larger than the size supported by any implementation that supports the Data Address Compare mechanism, or KEY0:1) and PTE152:53 (which are either reserved bits or KEY2:3) to 0s.
- Use PTEKEY values 0 and 1 only for purposes of emulating the Data Address Compare mechanism, except that PTEKEY value 0 may also be used for any virtual pages for which it is desired that the Virtual Page Class Key Protection mechanism permit all accesses. Do not use PTEKEY =31.
- When a Hypervisor Data Storage interrupt occurs, if HDSISR42=1 then ignore the interrupt for Cache Management instructions other than dcbz. (These instructions can cause a virtual page class key protection violation but cannot cause a Data Address Compare match.) Otherwise forward the interrupt to the operating system, which will treat the interrupt as if a Data Address Compare match had occurred. (Note: Cases for which it is undefined whether a Data Address Compare match occurs do not necessarily cause a virtual page class key protection violation.)

(Because privileged software can access the AMR using either SPR 13 or SPR 29, it might seem that, when SPR 13 was added to the architecture (in Version 2.06), SPR 29 should have been removed. SPR 29 is retained for two reasons: first, to avoid requiring privileged software to change to use the newer SPR number; and second, to retain the ability to emulate the Data Address Compare mechanism as described above.)
An example of the use of the AMOR (and UAMOR) is to support lightweight partitions, here called "adjunct" partitions, that provide services (e.g., device drivers) to "client" partitions. The adjunct partition would be managed by the hypervisor. It would run in problem state with MSR$$_{HV}$$=0b11, thereby restricting the resources it can modify (MSR$$_{PR}$$=1) and causing its interrupts to go to the hypervisor (MSR$$_{HV}$$=1), and it would share a Page Table with the client partition it serves. Typically, each of the two partitions would have data storage that the other partition must not be able to access. The hypervisor can use the AMOR, UAMOR, AMR, and PTE KEY field to provide the required protection. (The adjunct partition's lightness of weight derives from not requiring an operating system, and especially from not requiring a full partition context switch (SLB flush, TLB flush, SDR1 change, etc.) when the client partition invokes the services of the adjunct partition.)

For example, suppose each of the two partitions must not be able to access any of the other partition's data storage. The hypervisor could use KEY value j for all data virtual pages that only the adjunct partition must be able to access. Before dispatching the client partition for the first time, the hypervisor would initialize the three registers as follows.

AMR: all 0s except bits 2j and 2j+1, which would contain 1s
UAMOR: all 0s
AMOR: all 1s except bits 2j and 2j+1, which would contain 0s

Before dispatching the adjunct partition, the hypervisor would set UAMOR to all 0s, and would set the AMR to all 1s except bits 2j and 2j+1, which would be set to 0s. (Because the adjunct partition would run in problem state, there is no need for the hypervisor to modify the AMOR, and the adjunct partition cannot modify the UAMOR.) In addition, the hypervisor would prevent the client partition from modifying or deleting PTEs that contain translations used by the adjunct partition.

(It may be desirable to avoid using KEY values 0, 1, and 31 for storage that only the adjunct partition can access, because these KEY values may be needed by the client partition to emulate the Data Address Compare mechanism, as described above. Also, old software, that was written for an implementation that complies with a version of the architecture that precedes Version 2.04 (the version in which virtual page class keys were added), effectively uses KEY 0 for all virtual pages.)

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are "AMR-unaware". In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an "AMR-unaware" application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for "AMR-unaware" application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be "AMR-aware" — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an "AMR-unaware" program from reading the AMR, but inadvertently reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

(For partitions that run in a compatibility mode corresponding to a version of the architecture that precedes Version 2.06, the PCR provides sufficient protection to application programs.)

### Programming Note

An example of the use of the AMOR (and UAMOR) is to support lightweight partitions, here called "adjunct" partitions, that provide services (e.g., device drivers) to "client" partitions. The adjunct partition would be managed by the hypervisor. It would run in problem state with MSR$$_{HV}$$=0b11, thereby restricting the resources it can modify (MSR$$_{PR}$$=1) and causing its interrupts to go to the hypervisor (MSR$$_{HV}$$=1), and it would share a Page Table with the client partition it serves. Typically, each of the two partitions would have data storage that the other partition must not be able to access. The hypervisor can use the AMOR, UAMOR, AMR, and PTE KEY field to provide the required protection. (The adjunct partition’s lightness of weight derives from not requiring an operating system, and especially from not requiring a full partition context switch (SLB flush, TLB flush, SDR1 change, etc.) when the client partition invokes the services of the adjunct partition.)

For example, suppose each of the two partitions must not be able to access any of the other partition's data storage. The hypervisor could use KEY value j for all data virtual pages that only the adjunct partition must be able to access. Before dispatching the client partition for the first time, the hypervisor would initialize the three registers as follows.

AMR: all 0s except bits 2j and 2j+1, which would contain 1s
UAMOR: all 0s
AMOR: all 1s except bits 2j and 2j+1, which would contain 0s

Before dispatching the adjunct partition, the hypervisor would set UAMOR to all 0s, and would set the AMR to all 1s except bits 2j and 2j+1, which would be set to 0s. (Because the adjunct partition would run in problem state, there is no need for the hypervisor to modify the AMOR, and the adjunct partition cannot modify the UAMOR.) In addition, the hypervisor would prevent the client partition from modifying or deleting PTEs that contain translations used by the adjunct partition.

(It may be desirable to avoid using KEY values 0, 1, and 31 for storage that only the adjunct partition can access, because these KEY values may be needed by the client partition to emulate the Data Address Compare mechanism, as described above. Also, old software, that was written for an implementation that complies with a version of the architecture that precedes Version 2.04 (the version in which virtual page class keys were added), effectively uses KEY 0 for all virtual pages.)

### Programming Note

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are "AMR-unaware". In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an "AMR-unaware" application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for "AMR-unaware" application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be "AMR-aware" — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an "AMR-unaware" program from reading the AMR, but inadvertently reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

(For partitions that run in a compatibility mode corresponding to a version of the architecture that precedes Version 2.06, the PCR provides sufficient protection to application programs.)

### Programming Note

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are "AMR-unaware". In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an "AMR-unaware" application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for "AMR-unaware" application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be "AMR-aware" — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an "AMR-unaware" program from reading the AMR, but inadvertently reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

(For partitions that run in a compatibility mode corresponding to a version of the architecture that precedes Version 2.06, the PCR provides sufficient protection to application programs.)
Kₚ and Kₛ, the supervisor (privileged) state and problem state storage key bits in the SLB entry used to translate the effective address.

PP, page protection bits 0:2 in the Page Table Entry used to translate the effective address.

- For instruction fetches only:
  - the N (No-execute) value used for the access (see Sections 5.7.6.1 and 5.7.7.3)
  - PTE_INDEX, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE_INDEX = 1.

2. For any access except an instruction fetch that is not permitted by rule 1, a “Key” value is computed using the following formula:

$$\text{Key} = (K_p \& \text{MSR PR}) | (K_s \& \neg \text{MSR PR})$$

Using the computed Key, Figure 32 is applied. An instruction fetch is permitted for any entry in the figure except “no access”. A load is permitted for any entry except “no access”. A store is permitted only for entries with “read/write”.

Using the above values, the following rules are applied.

1. If MSR_HV = 0 and VPM₀ = 1, access authority is determined as described in Section 5.7.3.4.

2. If MSR_HV = 1 or VPM₀ = 0, Figure 33 is applied. The access is permitted for any entry in the figure except “no access”.

<table>
<thead>
<tr>
<th>HV</th>
<th>Access Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read/write or no access¹</td>
</tr>
<tr>
<td>1</td>
<td>read/write</td>
</tr>
</tbody>
</table>

¹ If the effective address for the access is less than the value specified by the RMLS, the access authority is read/write; otherwise the access is not permitted.

Figure 33. Protection states, address translation disabled

Programming Note

The comparison described in note 1 in Figure 33 ignores bits 0:3 of the effective address and may ignore bits 4:63-m; see Section 5.7.3.

### 5.7.9.3 Basic Storage Protection, Address Translation Disabled

When address translation is disabled, the Basic Storage Protection mechanism is controlled by the following (see Chapter 2 and Section 5.7.3, “Real And Virtual Real Addressing Modes”).

- MSR_HV, which (when MSR_PP = 0) distinguishes between hypervisor state and privileged but non-hypervisor state
- VPM₀, which distinguishes between real addressing mode and virtual real addressing mode
- RMLS, which specifies the real mode limit value

<table>
<thead>
<tr>
<th>Key</th>
<th>PP</th>
<th>Access Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>read/write</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>read/write</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>read/write</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>read only</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>read only</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>no access</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>read only</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>read/write</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>read only</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>no access</td>
</tr>
</tbody>
</table>

All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.

Figure 32. PP bit protection states, address translation enabled
5.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

5.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if any of the following conditions is satisfied.

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, and the G bit is 1 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR_{HV}=1, and the storage is outside the range(s) specified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.3.1).

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of Load and Store instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- Load or Store instruction that causes an atomic access
  - If any portion of the storage operand has been accessed and an External, Decrementer, Hypervisor Decrementer, Performance Monitor, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.
- Load or Store instruction that causes an Alignment exception, or that causes a [Hypervisor] Data Storage exception for reasons other than Data Address Watchpoint match.

The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

5.8.2 Storage Control Bits

When address translation is enabled, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corre-
sponding storage control for all accesses translated by the entry as shown in Figure 34.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Storage Control Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W^1,3$</td>
<td>0 - not Write Through Required</td>
</tr>
<tr>
<td>I$^3$</td>
<td>0 - not Caching Inhibited</td>
</tr>
<tr>
<td>M$^2$</td>
<td>0 - not Memory Coherence Required</td>
</tr>
<tr>
<td>G</td>
<td>0 - not Guarded</td>
</tr>
</tbody>
</table>

1 Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.

2 [Category: Memory Coherence] Support for the 0 value of the M bit is optional, implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

3 [Category: SAO] The combination $\text{WIMG} = \text{0b1110}$ has behavior unrelated to the meanings of the individual bits. See see Section 5.8.2.1, “Storage Control Bit Restrictions” for additional information.

Figure 34. Storage control bits

When address translation is enabled, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1; see Section 5.7.9.

When address translation is disabled, the storage control attributes are implicit; see Section 5.7.3.3.

In Sections 5.8.2.1 and 5.8.2.2, “access” includes accesses that are performed out-of-order, and references to $W$, $I$, $M$, and $G$ bits include the values of those bits that are implied when address translation is disabled.

Programming Note

In a system consisting of only a single-threaded processor which has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

5.8.2.1 Storage Control Bit Restrictions

All combinations of $W$, $I$, $M$, and $G$ values are permitted except those for which both $W$ and $I$ are 1 and $MI/G \neq \text{0b10}$.

The combination $\text{WIMG} = \text{0b1110}$ is used to identify the Strong Access Ordering (SAO) storage attribute (see Section 1.7.1, “Storage Access Ordering”, in Book II). Because this attribute is not intended for general purpose programming, it is provided only for a single combination of the attributes normally identified using the WIMG bits. That combination would normally be indicated by $\text{WIMG} = \text{0b0010}$.

References to Caching Inhibited storage (or storage with $I=1$) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the encoding using $I=1$. Conversely, references to storage that is not Caching Inhibited (or storage with $I=0$) apply to SAO storage or its WIMG encoding. References to Write Through Required storage (or storage with $W=1$) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses $W=1$. Conversely, references to storage that is not Write Through Required (or storage with $W=0$) apply to SAO storage or its WIMG encoding.

If a given real page is accessed concurrently as SAO storage and as non-SAO storage, the result may be characteristic of the weakly consistent model.

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0. For implementations that support the SAO category, the operating system should provide a means by which application programs can request SAO storage, in order to avoid confusion with the preceding guideline (since SAO is encoded using $WI=\text{0b11}$).

At any given time, the value of the W bit must be the same for all accesses to a given real page.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

5.8.2.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using $\text{dcbst}$ or $\text{dcbf}[l]$.

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using $\text{dcbf}[l]$ and $\text{idcbi}$ before permitting any other accesses to the page. Note that similar cache management is required before using the Fixed-Point Load and Store Caching Inhibited instructions to
access storage that has formerly been cached. (See Section 4.4.1 on page 875.)

--- Programming Note ---

The storage control bit alterations described above are examples of cases in which the directives for application of statements about the W and I bits to SAO given in the third paragraph of the preceding subsection must be applied. A transition from the typical WIMG=0b0010 for ordinary storage to WIMG=0b1110 for SAO storage does not require the flush described above because both WIMG combinations indicate storage that is not Caching Inhibited.

--- Programming Note ---

It is recommended that \texttt{dcbf} be used, rather than \texttt{dcbfl}, when changing the value of the I or W bit from 0 to 1. (\texttt{dcbfl} would have to be executed on all threads for which the contents of the data cache may be inconsistent with the new value of the bit, whereas, if the M bit for the page is 1, \texttt{dcbf} need be executed on only one thread in the system.)

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this are system-dependent.

--- Programming Note ---

For example, when changing the M bit in some directory-based systems, software may be required to execute \texttt{dcbf} on each thread to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

Additional requirements for changing the storage control bits in the Page Table are given in Section 5.10.
5.9 Storage Control Instructions

5.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a `dcbz` instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 5.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not maintained.

5.9.2 Synchronize Instruction

The `Synchronize` instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers.

When L=0, the `sync` instruction also provides an ordering function for the operations caused by the `Message Send` instruction and previous `Stores`. The stores must be performed with respect to the processor receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt occurs.

When L=1, the `sync` instruction provides an ordering function for the operations caused by the `Message Send` instruction and previous `Stores` for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The stores must be performed with respect to the processor receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt occurs.

Another variant of the `Synchronize` instruction is described below. It is designated the Page Table Entry Synchronize instruction, and is specified by the extended mnemonic `ptesync` (equivalent to `sync` with L=2).

The `ptesync` instruction has all of the properties of `sync` with L=0 and also the following additional properties.

- The memory barrier created by the `ptesync` instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the thread executing the `ptesync` instruction and, as elements of set A, for all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the `ptesync` instruction, before the `ptesync` instruction is executed. The applicable pairs are all pairs \(a_i,b_j\) in which \(b_j\) is a data access and \(a_i\) is not an instruction fetch.

- The `ptesync` instruction causes all Reference and Change bit updates associated with address translations that were performed, by the thread executing the `ptesync` instruction, before the `ptesync` instruction is executed, to be performed with respect to that thread before the `ptesync` instruction's memory barrier is created.

- The `ptesync` instruction provides an ordering function for all stores to the Page Table caused by `Store` instructions preceding the `ptesync` instruction with respect to searches of the Page Table that are performed, by the thread executing the `ptesync` instruction, after the `ptesync` instruction completes. Executing a `ptesync` instruction ensures that all such stores will be performed, with respect to the thread executing the `ptesync` instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table searches, are performed with respect to that thread.

- In conjunction with the `tlbie` and `tlbsync` instructions, the `ptesync` instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the `tlbsync` instruction description on page 934.
5.9.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB). For performance reasons, most implementations also have implementation-specific lookaside information that is used in address translation. This lookaside information may be: a Translation Lookaside Buffer (TLB) which is a cache of recently used Page Table Entries (PTEs); a cache of recently used translations of effective addresses to real addresses; etc.; or any combination of these. Lookaside information, including the SLB, is managed using the instructions described in the subsections of this section.

Lookaside information derived from PTEs is not necessarily kept consistent with the Page Table. When software alters the contents of a PTE, in general it must also invalidate all corresponding implementation-specific lookaside information; exceptions to this rule are described in Section 5.10.1.2.

The effects of the `slbie`, `slbia`, and TLB Management instructions on address translations, as specified in Sections 5.9.3.1 and 5.9.3.3 for the SLB and TLB respectively, apply to all implementation-specific lookaside information that is used in address translation. Unless otherwise stated or obvious from context, references to SLB entry invalidation and TLB entry invalidation elsewhere in the Books apply also to all implementation-specific lookaside information that is derived from SLB entries and PTEs respectively.

The `tlbia` instruction is optional. However, all implementations provide a means by which software can invalidate all implementation-specific lookaside information that is derived from PTEs.

Implementation-specific lookaside information that contains translations of effective addresses to real addresses may include “translations” that apply in real addressing mode. Because such “translations” are affected by the contents of the LPCR, RMOR, and HRMOR, when software alters the contents of these registers it must also invalidate the corresponding implementation-specific lookaside information. Software can invalidate all such lookaside information by using the `slbia` instruction with IH=0b000. However, performance is likely to be better if other, appropriate, IH values are used to limit the amount of lookaside information that invalidated.

All implementations that have such lookaside information provide a means by which software can invalidate all such lookaside information.

For simplicity, elsewhere in the Books it is assumed that the TLB exists.

--- Programming Note ---

Because the instructions used to manage implementation-specific lookaside information that is derived from PTEs may be changed in a future version of the architecture, it is recommended that software “encapsulate” uses of the TLB Management instructions into subroutines.

--- Programming Note ---

The function of all the instructions described in Sections 5.9.3.1 - 5.9.3.3 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 12.

5.9.3.1 SLB Management Instructions

--- Programming Note ---

Accesses to a given SLB entry caused by the instructions described in this section obey the sequential execution model with respect to the contents of the entry and with respect to data dependencies on those contents. That is, if an instruction sequence contains two or more of these instructions, when the sequence has completed, the final contents of the SLB entry and of General Purpose Registers is as if the instructions had been executed in program order.

However, software synchronization is required in order to ensure that any alterations of the entry take effect correctly with respect to address translation; see Chapter 12.
Changes to the segment mappings in the presence of active transactions may compromise transactional semantics if the transaction has accessed a segment that is assigned a new VSID. Consequently, when modifying segment mappings, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified segment is terminated, using a `tabort` or `treclaim` instruction.

---

**Programming Note**

Changes to the segment mappings in the presence of active transactions may compromise transactional semantics if the transaction has accessed a segment that is assigned a new VSID. Consequently, when modifying segment mappings, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified segment is terminated, using a `tabort` or `treclaim` instruction.

---

### SLB Invalidate Entry

<table>
<thead>
<tr>
<th>slbie</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>434</td>
</tr>
</tbody>
</table>

\[
ea_{0:35} \leftarrow \langle RB \rangle_{0:35}
\]

if, for SLB entry that translates
or most recently translated ea,
entry_class = \langle RB \rangle_{36}
entry_seg_size = size specified in \langle RB \rangle_{37:38}
then for SLB entry (if any) that translates ea
SLBE_V \leftarrow 0
all other fields of SLBE \leftarrow undefined
else
s \leftarrow \log_{2}(\text{entry_seg_size})
esid \leftarrow \langle RB \rangle_{0:63} - s
u \leftarrow undefined 1-bit value
if u then
if an SLB entry translates esid
SLBE_V \leftarrow 0
all other fields of SLBE \leftarrow undefined

Let the Effective Address (EA) be any EA for which \(EA_{0:35} = \langle RB \rangle_{0:35}\). Let the class be \(\langle RB \rangle_{36}\). Let the segment size be equal to the segment size specified in \(\langle RB \rangle_{37:38}\); the allowed values of \(\langle RB \rangle_{37:38}\), and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 21 on page 897).

The class value and segment size must be the same as the class value and segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address “translation” is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

The hardware ignores the contents of RB listed below and software must set them to 0s.

- \(\langle RB \rangle_{37}\)
- \(\langle RB \rangle_{39}\)
- \(\langle RB \rangle_{40:63}\)
- If \(s = 40\), \(\langle RB \rangle_{24:35}\)
If this instruction is executed in 32-bit mode, \((RB)_{0:31}\) must be zeros.

This instruction is privileged.

**Special Registers Altered:**

None

---

**Programming Note**

*slbie* does not affect SLBs on other threads.

---

**Programming Note**

The reason the class value specified by *slbie* must be the same as the Class value that is or was in the relevant SLB entry is that the hardware may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by *slbie* differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some implementations maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.)

When switching tasks in certain cases, it may be advantageous to preserve some implementation-specific lookaside entries while invalidating others. The IH=0b001 invalidation hint of the *slbia* instruction can be used for this purpose if SLB class values are appropriately assigned, i.e., a class value of 0 gives the hint that the entry should be preserved and a class value of 1 indicates the entry must be invalidated. Also, it is advantageous to assign a class value of 1 to entries that need to be invalidated via an *slbie* instruction while preserving implementation-specific lookaside entries that are not derived from an SLB entry since such entries are assigned a class value of 0.

The *Move To Segment Register* instructions (see Section 5.9.3.2.1) create SLB entries in which the Class value is 0.

---

**Programming Note**

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

---

### SLB Invalidate All

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>slbia</td>
</tr>
</tbody>
</table>

For each SLB entry except SLB entry 0

\[
\text{SLBE}_v \leftarrow 0
\]

all other fields of SLBE \leftarrow undefined

For all SLB entries except SLB entry 0, the V bit in the entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values. SLB entry 0 is not altered.

On implementations that have implementation-specific lookaside information for effective to real address translations, the IH field provides a hint that can be used to invalidate entries selectively in such lookaside information. The defined values for IH are as follows.

- 0b000 All such implementation-specific lookaside information is invalidated. (This value is not a hint.)
- 0b001 Preserve such implementation-specific lookaside information having a Class value of 0.
- 0b010 Preserve such implementation-specific lookaside information created when MSRIR/DR=0.
- 0b110 Preserve such implementation-specific lookaside information created when MSRIR/DR=0.

All other IH values are reserved. If the IH field contains a reserved value, the hint provided by the IH field is undefined.

Implementation specific lookaside information for which preservation is not requested is invalidated. Implementation specific lookaside information for which preservation is requested may be invalidated.

When IH=0b000, execution of this instruction has the side effect of clearing the storage access history associated with the Hypervisor Real Mode Storage Control facility. See Section 5.7.3.3.1, “Hypervisor Real Mode Storage Control” for more details.

This instruction is privileged.

**Special Registers Altered:**

None

---

**Programming Note**

*slbia* does not affect SLBs on other threads.
Programming Note

If `slbia` is executed when instruction address translation is enabled, software can ensure that attempting to fetch the instruction following the `slbia` does not cause an Instruction Segment interrupt by placing the `slbia` and the subsequent instruction in the effective segment mapped by SLB entry 0. (The preceding assumes that no other interrupts occur between executing the `slbia` and executing the subsequent instruction.)

If it is desired to invalidate the entire SLB and all associated implementation-specific lookaside information, the following sequence can be used. The sequence assumes that address translation is disabled.

```
li r0,0
slbmt r0,r0 # clear SLBE 0
slbia 0b0000 # invalidate all other
# SLBEs, and all ERATEs
```

Programming Note

The defined values for IH are as follows.

<table>
<thead>
<tr>
<th>0b000</th>
<th>All ERAT entries are invalidated. (This value is not a hint.) This value should be used by the hypervisor when relocating itself (i.e. when modifying the HRMOR) or when reconfiguring real storage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b001</td>
<td>Preserve ERAT entries with a Class value of 0. This value should be used by an operating system when switching tasks in certain cases; for example, if SLBE&lt;sub&gt;C&lt;/sub&gt;=0 is used for SLB translations shared between the tasks.</td>
</tr>
<tr>
<td>0b010</td>
<td>Preserve ERAT entries created when MSR&lt;sub&gt;IR&lt;/sub&gt;/DR=0. This value should generally be used by an operating system when switching tasks.</td>
</tr>
<tr>
<td>0b110</td>
<td>Preserve ERAT entries created when MSR&lt;sub&gt;HV&lt;/sub&gt;=1 and MSR&lt;sub&gt;IR&lt;/sub&gt;/DR=0. This value should be used by the hypervisor when switching partitions.</td>
</tr>
</tbody>
</table>

**SLB Move To Entry**

<table>
<thead>
<tr>
<th>X-form</th>
<th>( \text{slbmt} )</th>
<th>RS, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31 ( \text{RS} )</td>
<td></td>
</tr>
</tbody>
</table>

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 35.

**RS**

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>K&lt;sub&gt;s&lt;/sub&gt;K&lt;sub&gt;p&lt;/sub&gt;NLC</th>
<th>0</th>
<th>LP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>57</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>

**RB**

<table>
<thead>
<tr>
<th>ESID</th>
<th>V</th>
<th>0s</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>37</td>
<td>52</td>
</tr>
</tbody>
</table>

**Programming Note**

`slbia` serves as both a basic and an extended mnemonic. The Assembler will recognize an `slbia` mnemonic with one operand as the basic form, and an `slbia` mnemonic with no operand as the extended form. In the extended form the IH operand is omitted and assumed to be 0.

**Programming Note**

If `slbia` is executed when instruction address translation is enabled, software can ensure that attempting to fetch the instruction following the `slbia` does not cause an Instruction Segment interrupt by placing the `slbia` and the subsequent instruction in the effective segment mapped by SLB entry 0. (The preceding assumes that no other interrupts occur between executing the `slbia` and executing the subsequent instruction.)

If it is desired to invalidate the entire SLB and all associated implementation-specific lookaside information, the following sequence can be used. The sequence assumes that address translation is disabled.

```
li r0,0
slbmt r0,r0 # clear SLBE 0
slbia 0b0000 # invalidate all other
# SLBEs, and all ERATEs
```

**Figure 35. GPR contents for slbmt**

On implementations that support a virtual address size of only \( n \) bits, \( n < 78 \), (RS)<sub>2:79-n</sub> must be zeros.

(RS)<sub>57</sub> and (RS)<sub>60:63</sub> are ignored by the hardware.

High-order bits of (RB)<sub>52:63</sub> that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

If this instruction is executed in 32-bit mode, (RB)<sub>0:31</sub> must be zeros (i.e., the ESID must be in the range 0:15).

This instruction cannot be used to invalidate the translation contained in an SLB entry.

**Special Registers Altered:**

None
SLB Move From Entry VSID  X-form

<table>
<thead>
<tr>
<th>slbmfsv</th>
<th>RT, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>///</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>851</td>
<td>31</td>
</tr>
</tbody>
</table>

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the B, VSID, K_s, K_p, N, L, C, and LP fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 36.

RT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>21</td>
<td>851</td>
</tr>
<tr>
<td>RT</td>
<td>RB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RB

<table>
<thead>
<tr>
<th>Os</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
</tr>
<tr>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

RT_{0:1} B
RT_{2:51} VSID
RT_{52} K_s
RT_{53} K_p
RT_{54} N
RT_{55} L
RT_{56} C
RT_{57} set to 0b0
RT_{58:59} LP
RT_{60:63} set to 0b0000

RB_{0:51} must be 0x0_0000_0000_0000
RB_{52:63} index, which selects the SLB entry

Figure 36. GPR contents for slbmfsv

On implementations that support a virtual address size of only n bits, n<78, RT_{2:79-n} are set to zeros.

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of (RB)_{52:63} that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

Special Registers Altered:
None
**SLB Move From Entry ESID**

\[ \text{slbmmfee RT, RB} \]

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 37.

**RT**

<table>
<thead>
<tr>
<th>ESID</th>
<th>V</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>37</td>
</tr>
</tbody>
</table>

**RB**

<table>
<thead>
<tr>
<th>0s</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
</tr>
<tr>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

- \( RT_{0:35} \) ESID
- \( RT_{36} \) V
- \( RT_{37:63} \) set to 0b000 || 0x0_00_0000_0000
- \( RB_{0:51} \) must be 0x0_00_0000_0000
- \( RB_{52:63} \) index, which selects the SLB entry

**Figure 37. GPR contents for slbmmfee**

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of \( RB_{52:63} \) that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

**Special Registers Altered:** None

**SLB Find Entry ESID**

\[ \text{slbfee RT, RB} \]

The SLB is searched for an entry that matches the effective address specified by register RB. The search is performed as if it were being performed for purposes of address translation. That is, in order for a given entry to satisfy the search, the entry must be valid (V=1), and \( (RB)_{63-s} \) must equal \( \text{SLBE}[	ext{ESID}_{63-s}] \) (where \( 2^s \) is the segment size selected by the B field in the entry). If exactly one matching entry is found, the contents of the B, VSID, \( K_p, K_p, N, L, C, \) and LP fields of the entry are placed into register RT. If no matching entry is found, register RT is set to 0. If more than one matching entry is found, either one of the matching entries is used, as if it were the only matching entry, or a Machine Check occurs. If a Machine Check occurs, register RT, and CR Field 0 are set to undefined values, and the description below of how this register and this field is set does not apply.

The contents of registers RT and RB are interpreted as shown in Figure 38.

**RT**

<table>
<thead>
<tr>
<th>B</th>
<th>VSID</th>
<th>( K_p, K_p, N, L, C )</th>
<th>0</th>
<th>LP</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>57</td>
<td>58</td>
<td>60</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RB**

<table>
<thead>
<tr>
<th>ESID</th>
<th>0000</th>
<th>0s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( RT_{0:1} \) B
- \( RT_{2:51} \) VSID
- \( RT_{52} \) \( K_p \)
- \( RT_{53} \) \( K_p \)
- \( RT_{54} \) N
- \( RT_{55} \) L
- \( RT_{56} \) C
- \( RT_{57} \) set to 0b0
- \( RT_{58:59} \) LP
- \( RT_{60:63} \) set to 0b0000
- \( RB_{0:35} \) ESID
- \( RB_{36:39} \) must be 0b0000
- \( RB_{40:63} \) must be 0x0000000

**Figure 38. GPR contents for slbfee.**

If \( s > 28 \), \( RT_{80-s:51} \) are set to zeros. On implementations that support a virtual address size of only \( n \) bits, \( n < 78 \), \( RT_{7:79-n} \) are set to zeros.

CR Field 0 is set as follows. \( j \) is a 1-bit value that is equal to 0b1 if a matching entry was found. Otherwise, \( j \) is 0b0.

\[
\text{CR0}_{LT} \text{GT} \text{EQ} \text{SO} = 0b00 \| j \| \text{XER}_{SO}
\]
If this instruction is executed in 32-bit mode, \((RB)_{0:31}\) must be zeros (i.e., the ESID must be in the range 0-15).

This instruction is privileged.

**Special Registers Altered:**
- CR0
5.9.3.2 Bridge to SLB Architecture [Category:Server.Phased-Out]

The facility described in this section can be used to ease the transition to the current Power ISA software-managed Segment Lookaside Buffer (SLB) architecture, from the Segment Register architecture provided by 32-bit PowerPC implementations. A complete description of the Segment Register architecture may be found in “Segmented Address Translation, 32-Bit Implementations,” Section 4.5, Book III of Version 1.10 of the PowerPC architecture, referenced in the introduction to this architecture.

The facility permits the operating system to continue to use the 32-bit PowerPC implementation’s Segment Register Manipulation instructions.

5.9.3.2.1 Segment Register Manipulation Instructions

The instructions described in this section -- mtssr, mtssrin, msfr, and mfsrin -- allow software to associate effective segments 0 through 15 with any of virtual segments 0 through 227-1. SLB entries 0:15 serve as virtual Segment Registers, with SLB entry i used to emulate Segment Register i. The mtssr and mtssrin instructions move 32 bits from a selected GPR to a selected SLB entry. The msfr and mfsrin instructions move 32 bits from a selected SLB entry to a selected GPR.

The contents of the GPRs used by the instructions described in this section are shown in Figure 39. Fields shown as zeros must be zero for the Move To Segment Register instructions. Fields shown as hyphens are ignored. Fields shown as periods are ignored by the Move To Segment Register instructions and set to undefined values by the Move From Segment Register instructions. Fields shown as colons are ignored by the Move To Segment Register instructions and set to undefined values by the Move From Segment Register instructions.

<table>
<thead>
<tr>
<th>RS/RT</th>
<th>: : :</th>
<th>K</th>
<th>K</th>
<th>N</th>
<th>0</th>
<th>VSID</th>
<th>23:49</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>32</td>
<td>33</td>
<td>36</td>
<td>37</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RB</th>
<th>- - -</th>
<th>ESID</th>
<th>- - -</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>36</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 39. GPR contents for mtssr, mtssrin, msfr, and mfsrin

The “Segment Register” format used by the instructions described in this section corresponds to the low-order 32 bits of RS and RT shown in the figure. This format is essentially the same as that for the Segment Registers of 32-bit PowerPC implementations. The only differences are the following.

- Bit 36 corresponds to a reserved bit in Segment Registers. Software must supply 0 for the bit because it corresponds to the L bit in SLB entries, and large pages are not supported for SLB entries created by the Move To Segment Register instructions.
- VSID bits 23:25 correspond to reserved bits in Segment Registers. Software can use these extra VSID bits to create VSIDs that are larger than those supported by the Segment Register Manipulation instructions of 32-bit PowerPC implementations.

Bit 32 of RS and RT corresponds to the T (direct-store) bit of early 32-bit PowerPC implementations. No corresponding bit exists in SLB entries.

Programming Note

The Programming Note in the introduction to Section 5.9.3.1 applies also to the Segment Register Manipulation instructions described in this section, and to any combination of the instructions described in the two sections, except as specified below for msfr and mfsrin.

The requirement that the SLB contain at most one entry that translates a given effective address (see Section 5.7.6.1) applies to SLB entries created by mtssr and mtssrin. This requirement is satisfied naturally if only mtssr and mtssrin are used to create SLB entries for a given ESID, because for these instructions the association between SLB entries and ESID values is fixed (SLB entry i is used for ESID i). However, care must be taken if slbmt is also used to create SLB entries for the ESID, because for slbmt the association between SLB entries and ESID values is specified by software.
Move To Segment Register X-form

mtsr SR,RS

The SLB entry specified by SR is loaded from register RS, as follows.

<table>
<thead>
<tr>
<th>SLBE Bit(s)</th>
<th>Set to</th>
<th>SLB Field(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>0x0000_0000</td>
<td>ESID0:31</td>
</tr>
<tr>
<td>32:35</td>
<td>SR</td>
<td>ESID32:35</td>
</tr>
<tr>
<td>36</td>
<td>0b1</td>
<td>V</td>
</tr>
<tr>
<td>37:38</td>
<td>0b00</td>
<td>B</td>
</tr>
<tr>
<td>39:61</td>
<td>0b000000000</td>
<td>VSID0:22</td>
</tr>
<tr>
<td>62:88</td>
<td>(RS)37:63</td>
<td>VSID23:49</td>
</tr>
<tr>
<td>89:91</td>
<td>(RS)33:35</td>
<td>KpKpN</td>
</tr>
<tr>
<td>92</td>
<td>(RS)36</td>
<td>L ((RS)36 must be 0b0)</td>
</tr>
<tr>
<td>93</td>
<td>0b0</td>
<td>C</td>
</tr>
<tr>
<td>94</td>
<td>0b0</td>
<td>reserved</td>
</tr>
<tr>
<td>95:96</td>
<td>0b00</td>
<td>LP</td>
</tr>
</tbody>
</table>

MSR_SF must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:
None

Move To Segment Register Indirect X-form

mtsrin RS,RB

The SLB entry specified by (RB)32:35 is loaded from register RS, as follows.

<table>
<thead>
<tr>
<th>SLBE Bit(s)</th>
<th>Set to</th>
<th>SLB Field(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>0x0000_0000</td>
<td>ESID0:31</td>
</tr>
<tr>
<td>32:35</td>
<td>(RB)32:35</td>
<td>ESID32:35</td>
</tr>
<tr>
<td>36</td>
<td>0b1</td>
<td>V</td>
</tr>
<tr>
<td>37:38</td>
<td>0b00</td>
<td>B</td>
</tr>
<tr>
<td>39:61</td>
<td>0b000000000</td>
<td>VSID0:22</td>
</tr>
<tr>
<td>62:88</td>
<td>(RS)37:63</td>
<td>VSID23:49</td>
</tr>
<tr>
<td>89:91</td>
<td>(RS)33:35</td>
<td>KpKpN</td>
</tr>
<tr>
<td>92</td>
<td>(RS)36</td>
<td>L ((RS)36 must be 0b0)</td>
</tr>
<tr>
<td>93</td>
<td>0b0</td>
<td>C</td>
</tr>
<tr>
<td>94</td>
<td>0b0</td>
<td>reserved</td>
</tr>
<tr>
<td>95:96</td>
<td>0b00</td>
<td>LP</td>
</tr>
</tbody>
</table>

MSR_SF must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:
None
Move From Segment Register  X-form

mfsr   RT,SR

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>SR</th>
<th>///</th>
<th>595</th>
<th>/</th>
</tr>
</thead>
</table>

The contents of the low-order 27 bits of the VSID field and the contents of the K_s, K_p, N, and L fields of the SLB entry specified by SR are placed into register RT as follows.

SLBE Bit(s) Copied to          SLB Field(s)
62:88  RT_{37:63}  VSID_{23:49}
89:91  RT_{33:35}  K_sK_pN
92    RT_{36}   L (SLBE_L must be 0b0)

RT_{32} is set to 0. The contents of RT_{0:31} are undefined.

MSR_SF must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by mtsr or mtsrin and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). If the SLB entry is invalid (V=0), RT_{33:63} are set to 0. Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:
None

Move From Segment Register Indirect  X-form

mfsrin  RT,RB

<table>
<thead>
<tr>
<th>31</th>
<th>RT</th>
<th>///</th>
<th>RB</th>
<th>659</th>
<th>/</th>
</tr>
</thead>
</table>

The contents of the low-order 27 bits of the VSID field and the contents of the K_s, K_p, N, and L fields of the SLB entry specified by (RB)_{32:35} are placed into register RT as follows.

SLBE Bit(s) Copied to          SLB Field(s)
62:88  RT_{37:63}  VSID_{23:49}
89:91  RT_{33:35}  K_sK_pN
92    RT_{36}   L (SLBE_L must be 0b0)

RT_{32} is set to 0. The contents of RT_{0:31} are undefined.

MSR_SF must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by mtsr or mtsrin and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). If the SLB entry is invalid (V=0), RT_{33:63} are set to 0. Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:
None
5.9.3.3 TLB Management Instructions

**Programming Note**

Changes to the page table in the presence of active transactions may compromise transactional semantics if a page accessed by a translation is remapped within the lifetime of the transaction. Through the use of a *tlbie* instruction to the unmapped page, an operating system or hypervisor can ensure that any transaction that has touched the affected page is terminated.

Changes to local translation lookaside buffers, through the *tlbia* and *tlbiel* instructions have no effect on transactions. Consequently, if these instructions are used to invalidate TLB entries after the unmapping of a page, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified page is terminated, using a tabort. or treclaim instruction.

---

**TLB Invalidate Entry**

* X-form

```
**tlbie**    RB,RS
```

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>RS</th>
<th>RB</th>
<th>306</th>
<th>31</th>
</tr>
</thead>
</table>
| L | RB | if L = 0
  | then
  | base_pg_size = 4K
  | actual_pg_size = page size specified in (RB)56:58
  | i = 51
  | else
  | base_pg_size = base page size specified in (RB)44:51
  | actual_pg_size = actual page size specified in (RB)44:51
  | i = max(min(43,63-b),63-p)
| b | log_base_2(base_pg_size)
| p | log_base_2(actual_pg_size)
| sg_size | segment size specified in (RB)54:55

for each thread

for each TLB entry

if ((entry_VA14:i+14 = (RB)0:i) &
    (entry_sg_size = sg_size) &
    (entry_actualPg_size = actual_pg_size) &
    ((TLBE contains LPID &
      (TLBE_LPID = (RS)32:63) ) |
    (TLBEs do not contain LPID &
      (LPIDR_LPID = (RS)32:63) ) )
then
  if ((L = 0) | (b >= 20))
  then
    TLB entry ← invalid
  else
    if (entry_VA58:77-b = (RB)56:75-b)

Changes to local translation lookaside buffers, through the use of the *tlbie* instruction to the unmapped page, an operating system or hypervisor can ensure that any transaction that has touched the affected page is terminated.

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below, where L is (RB)63.

**RS**:

<table>
<thead>
<tr>
<th>0s</th>
<th>LPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>63</td>
<td>63</td>
</tr>
</tbody>
</table>

**RB** if L=0:

<table>
<thead>
<tr>
<th>AVA</th>
<th>0s</th>
<th>B</th>
<th>AP</th>
<th>0s</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>59</td>
<td>63</td>
</tr>
</tbody>
</table>

**RB** if L=1:

<table>
<thead>
<tr>
<th>AVA</th>
<th>LP</th>
<th>0s</th>
<th>B</th>
<th>AVAL</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>63</td>
</tr>
</tbody>
</table>

**RS** contains an LPID value. The supported (RS)32:63 values are the same as the LPID values supported in LPIDR. RS0:31 must contain zeros and are ignored by the hardware.

If the L field in RB contains 0, the base page size is 4 KB and RB56:58 (AP - Actual Page size field) must be set to the SLBE_LILP encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to log₂ (actual page size specified by (RB)56:58). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between (RB)44:51 (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_LP and the base page size and actual page size, except for the "r" bits (see Section 5.7.7.1 on page 900 and Figure 25 on page 901). Thus, b is equal to log₂ (base page size specified by (RB)44:51) and p is equal to log₂ (actual page size specified by (RB)44:51). Specifically, (RB)44+c:51 must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the maximum of 0 and (20-p).
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). (RB)log₂ contains bits 14:(i+14) of the virtual address translated by the TLB entry to be invalidated. If b>20, RB64-b:43
may contain any value and are ignored by the hardware.

- If \( b < 20 \), \((RB)_{56:75-b}\) must contain bits \(58:77-b\) of the virtual address translated by the TLB to be invalidated, and other bits in \((RB)_{56:62}\) may contain any value and are ignored by the hardware.

- If \( b \geq 20 \), \((RB)_{56:62}\) (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in \((RB)_{54:55}\) (B field). The contents of \(RB_{54:55}\) must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

\(RB_{52:53}\) and \(RB_{59:62}\) (when \((RB)_{63} = 0\)) must contain zeros and are ignored by the hardware.
All TLB entries on all threads that have all of the following properties are made invalid.

- The entry translates a virtual address for which all the following are true.
  - $V_{A14:14+i}$ is equal to $(RB)_{0:i}$.
  - $L=0$ or $b \geq 20$ or, if $L=1$ and $b=20$, $V_{A58:77-b}$ is equal to $(RB)_{56:75-b}$.
  - The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in $(RB)_{56:58}$.
  - The L field in RB is 1, the base page size of the entry matches the base page size specified in $(RB)_{44:51}$, and the actual page size of the entry matches the actual page size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The implementation’s TLB entries contain LPID values and TLBELLPID = $(RS)_{32:63}$.
  - The implementation’s TLB entries do not contain LPID values, and LPIDR LPID = $(RS)_{32:63}$. The LPIDR used for this comparison is in the same thread as the TLB entry being tested.

If the implementation’s TLB entries contain LPID values, additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches $(RS)_{32:63}$. If the implementation’s TLB entries do not contain LPID values, additional TLB entries may also be made invalid on any thread that is in the partition specified by $(RS)_{32:63}$.

MSR$_{SF}$ must be 1 when this instruction is executed; otherwise the results are undefined.

If the value specified in RS$_{32:63}$, RB$_{54:55}$, RB$_{56:58}$ (when RB$_{63}=0$), or RB$_{44:51}$ (when RB$_{63}=1$) is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

The operation performed by this instruction is ordered by the eieio (or sync or ptesync) instruction with respect to a subsequent tlbsync instruction executed by the thread executing the tlbie instruction. The operations caused by tlbie and tlbsync are ordered by eieio as a fourth set of operations, which is independent of the other three sets that eieio orders.

This instruction is hypervisor privileged.

See Section 5.10, “Page Table Update Synchronization Requirements” for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

- None

---

**Programming Note**

For tlbie[] instructions in which $(RB)_{63}=0$, the AP value in RB is provided to make it easier for the hardware to locate address translations, in lookaside buffers, corresponding to the address translation being invalidated.

For tlbie[] instructions the AP specification is not binary compatible with versions of the architecture that precede Version 2.06. As an example, for an actual page size of 64 KB AP=0b101, whereas software written for an implementation that complies with a version of the architecture that precedes V. 2.06 would have AP=100 since AP was a 1 bit value followed by Os in RB$_{57:58}$. If binary compatibility is important, for a 64 KB page software can use AP=0b101 on these earlier implementations since these implementations were required to ignore RB$_{57:58}$.

For tlbie[] instructions the AVA and AVAL fields in RB contain different VA bits from those in PTEAVA.
Chapter 5. Storage Control

TLB Invalidate Entry Local

TLB Invalidate Entry Local X-form

<table>
<thead>
<tr>
<th>IS</th>
<th>AVA</th>
<th>LP</th>
<th>IS</th>
<th>B</th>
<th>AVAL</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
<td>52</td>
<td>54</td>
<td>56</td>
<td>59</td>
<td>63</td>
</tr>
</tbody>
</table>

The Invalidation Selector (IS) field in RB has three defined values (0b00, 0b10, and 0b11). The IS value of 0b01 is reserved and is treated in the same manner as the corresponding case for instruction fields (see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” on page 5 in Book I).

**Engineering Note**

**IS field in RB contains 0b00**

If the L field in RB contains 0, the base page size is 4 KB and RB56:58 (AP - Actual Page size field) must be set to the SLBE_LLP encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to \( \log_2 \) (actual page size specified by RB56:58). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between (RB)44:51 (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_LP and the base page size and actual page size, except for the “r” bits (see Section 5.7.7.1 on page 900 and Figure 25 on page 901). Thus, b is equal to \( \log_2 \) (base page size specified by RB44:51) and p is equal to \( \log_2 \) (actual page size specified by RB44:51). Specifically, (RB)44+c:51 must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the maximum of 0 and (20-p).
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). (RB)0:i must contain bits 14:(i +14) of the virtual address translated by the TLB to be invalidated, and other bits in (RB)56:62 may...
contain any value and are ignored by the hardware.
- If \( b \geq 20 \), \((RB)_{56:62}\) (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in \((RB)_{54:55}\) (B field). The contents of \(RB_{54:55}\) must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

Let the segment size be equal to the segment size specified in \((RB)_{54:55}\) (B field). The contents of \(RB_{54:55}\) must be the same as the contents of PTEB used to create the TLB entry to be invalidated.

All TLB entries that have all of the following properties are made invalid on the thread executing the \texttt{tlbiel} instruction.
- The entry translates a virtual address for which all the following are true.
  - \( VA_{14:14+i} \) is equal to \((RB)_{0:i}\).
  - \( L=0 \) or \( b \geq 20 \) or, if \( L=1 \) and \( b < 20 \), \( VA_{58:77-b} \) is equal to \((RB)_{56:75-b}\).
- The segment size of the entry is the same as the segment size specified in \((RB)_{54:55}\).
- Either of the following is true:
  - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in \((RB)_{56:58}\).
  - The L field in RB is 1, the base page size of the entry matches the base page size specified in \((RB)_{44:51}\), and the actual page size of the entry matches the actual page size specified in \((RB)_{44:51}\).
- Either of the following is true:
  - The implementation’s TLB entries do not contain LPID values.
  - The implementation’s TLB entries contain LPID values and TLBE\(_{LPID} = \text{LPIDR}_{LPID}\).

**IS field in RB contains 0b10 or 0b11**

\((RB)_{i:51}\) (bits \(i=40:11\) of the SET field in \((RB)\)) specify a set of TLB entries, where \( i \) is an implementation-dependent value in the range 40:51. Each entry in the set is invalidated if any of the following conditions are met for the entry.
- The implementation’s TLB entries do not contain an LPID value.
- The IS field in RB contains 0b10 or MSR\(_{HV}=0\), the implementation’s TLB entries contain an LPID value, and TLBE\(_{LPID} = \text{LPIDR}_{LPID}\).
- The IS field in RB contains 0b11 and MSR\(_{HV}=1\).

How the TLB is divided into the \(52^i\) sets is implementation-dependent. The relationship of virtual addresses to these sets is also implementation-dependent. However, if, in an implementation, there can be multiple TLB entries for the same virtual address and same partition, then all these entries must be in a single set.

If the IS field in RB contains 0b10 or 0b11, it is implementation-dependent whether implementation-specific lookaside information that contains translations of effective addresses to real addresses is invalidated.

\(RB_{0:39}\) (when \((RB)_{52:53} = 0b10\) or 0b11), \(RB_{59:62}\) (when \((RB)_{52:53} = 0b00\) and \((RB)_{63}=0\)), and \(RB_{54:63}\) (when \((RB)_{52:53} = 0b10\) or 0b11) must contain 0s and are ignored by the hardware. When \(i>40\) and \((RB)_{52:53} = 0b10\) or 0b11, \(RB_{40:i-1}\) may contain any value and are ignored by the hardware.

Only TLB entries on the thread executing the \texttt{tlbiel} instruction are affected.

MSR\(_{SF}\) must be 1 when this instruction is executed; otherwise the results are boundedly undefined.

If the value specified in \((RB)_{54:55}\), \((RB)_{56:58}\) (when \((RB)_{52:53} = 0b00\) and \((RB)_{63} = 0\)), or \((RB)_{44:51}\) (when \((RB)_{52:53} = 0b00\) and \((RB)_{63} = 1\)) is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

This instruction is privileged.

See Section 5.10, “Page Table Update Synchronization Requirements” on page 935 for a description of other requirements associated with the use of this instruction.

**Special Registers Altered:**

None
Chapter 5. Storage Control

**Programming Note**

The primary use of this instruction by hypervisor software is to invalidate TLB entries prior to reassigning a thread to a new logical partition.

For IS = 0b10 or 0b11, it is implementation-dependent whether ERAT entries are invalidated. If the `tlbiel` instruction is being executed due to a partition swap, an `slbia` instruction can be used to invalidate the pertinent ERAT entries. If the `tlbiel` instruction is being executed to invalidate TLB entries with parity or ECC errors, the fact that the corresponding ERAT entries are not invalidated is immaterial. If the `tlbiel` instruction is being executed to invalidate multiple matching TLB entries, the fact that the corresponding ERAT entries are not invalidated is immaterial for implementations that never create multiple matching ERAT entries.

The primary use of this instruction by operating system software is to invalidate TLB entries that were created by the hypervisor using an implementation-specific hypervisor-managed TLB facility, if such a facility is provided.

`tlbiel` may be executed on a given thread even if the sequence `tlbie - eieio - tlbsync - ptesync` is concurrently being executed on another thread.

See also the Programming Notes with the description of the `tlbie` instruction.

---

### TLB Invalidate All

<table>
<thead>
<tr>
<th>X-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tlbia</code></td>
</tr>
</tbody>
</table>

![X-form Diagram](image)

**Programming Note**

`tlbia` does not affect TLBs on other threads.
The `tlbsync` instruction provides an ordering function for the effects of all `tlbiel` instructions executed by the thread executing the `tlbsync` instruction, with respect to the memory barrier created by a subsequent `pte-sync` instruction executed by the same thread. Executing a `tlbsync` instruction ensures that all of the following will occur.

- All TLB invalidations caused by `tlbiel` instructions preceding the `tlbsync` instruction will have completed on any other thread before any data accesses caused by instructions following the `pte-sync` instruction are performed with respect to that thread.

- All storage accesses by other threads for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed by other threads using the translations being invalidated, will have been performed with respect to the thread executing the `pte-sync` instruction, to the extent required by the associated Memory Coherence Required attributes, before the `pte-sync` instruction's memory barrier is created.

The operation performed by this instruction is ordered by the `eieio` (or `sync` or `pte-sync`) instruction with respect to preceding `tlbiel` instructions executed by the thread executing the `tlbsync` instruction. The operations caused by `tlbiel` and `tlbsync` are ordered by `eieio` as a fourth set of operations, which is independent of the other three sets that `eieio` orders.

The `tlbsync` instruction may complete before operations caused by `tlbiel` instructions preceding the `tlbsync` instruction have been performed.

This instruction is hypervisor privileged.

See Section 5.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

---

**Programming Note**

`tlbsync` should not be used to synchronize the completion of `tlbiel`.
5.10 Page Table Update Synchronization Requirements

This section describes rules that software must follow when updating the Page Table, and includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, the Page Table Entry is assumed to be for a virtual page for which the base page size is equal to the actual page size. If these page sizes are different, multiple tlbie instructions are needed, one for each PTE corresponding to the virtual page.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) that corresponds to a single line in the sequence is assumed to be done using a Store instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two Store Word instructions).

Stores are not performed out-of-order, as described in Section 5.5, “Performing Operations Out-of-Order” on page 890. Moreover, address translations associated with instructions preceding the corresponding Store instructions are not performed again after the stores have been performed. (These address translations must have been performed before the store was determined to be required by the sequential execution model, because they might have caused an exception.) As a result, an update to a PTE need not be preceded by a context synchronizing operation.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the PTE are to be used for address translations associated with subsequent instructions.

As noted in the description of the Synchronize instruction in Section 4.4.3 of Book II, address translation associated with instructions which occur in program order subsequent to the Synchronize (and this includes the ptesync variant) may be performed prior to the completion of the Synchronize. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

Programming Note

In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the rfid or hrfid instruction that returns from the interrupt handler may provide the required context synchronization.

Page Table Entries must not be changed in a manner that causes an implicit branch.

5.10.1 Page Table Updates

TLBs are non-coherent caches of the HTAB. TLB entries must be invalidated explicitly with one of the TLB Invalidate instructions.

Unsynchronized lookups in the HTAB continue even while it is being modified. Any thread, including a thread on which software is modifying the HTAB, may look in the HTAB at any time in an attempt to translate a virtual address. When modifying a PTE, software must ensure that the PTE’s V bit is 0 if the PTE is inconsistent (e.g., if the RPN field is not correct for the current AWA field).

Updates of Reference and Change bits by the hardware are not synchronized with the accesses that cause the updates. When modifying doubleword 1 of a PTE, software must take care to avoid overwriting a hardware update of these bits and to avoid having the value written by a Store instruction overwritten by a hardware update.

Software must execute tlbie and tlbsync instructions only as part of the following sequence, and must ensure that no other thread will execute a “conflicting instruction” while the instructions in the sequence are executing on the given thread. In addition to achieving the required system synchronization, the sequence will cause transactions that include accesses to the affected page(s) to fail.

- tlbie instruction(s) specifying the same LPID operand value
- eieio
- tlbsync
- ptesync

Let L be the LPID value specified by the above tlbie instruction(s). The “conflicting instructions” in this case are the following.

- a tlbie instruction that specifies an LPID value that matches the value L
- a tlbsync instruction that is part of a tlbie-eieio-tlbsync-ptesync sequence in which the tlbie instruction(s) specify an LPID value that matches the value L
- an mtspr instruction that modifies the LPIDR, if the modification has either of the following properties.
  - The old LPID value (i.e., the contents of the LPIDR just before the mtspr instruction is executed) is the value L
  - The new LPID value (i.e., the value specified by the mtspr instruction) is the value L

Other instructions (excluding mtspr instructions that modify the LPIDR as described above, and excluding...
**tlbie** instructions except as shown) may be interleaved with the instruction sequence shown above, but the instructions in the sequence must appear in the order shown. On systems consisting of only a single-threaded processor, the **eieio** and **tlbsync** instructions can be omitted.

---

**Programming Note**

The **eieio** instruction prevents the reordering of the preceding **tlbie** instructions with respect to the subsequent **tlbsync** instruction. The **tlbsync** instruction and the subsequent **ptesync** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated (by the **tlbie** instructions), and all Reference and Change bit updates associated with address translations that were performed using the translations being invalidated, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ptesync instruction are performed with respect to that thread or mechanism.

For Page Table update sequences that mark the PTE invalid (see Section 5.10.1.2, "Modifying a Page Table Entry"), Reference and Change bit updates can continue to be performed in the invalid PTE until the **ptesync** at the end of the **tlbie/eieio/tlbsync/ptesync** sequence has completed. Any access to the PTE, by software, that should be performed after all such implicit PTE updates have completed, such as reading the final values of the Reference and Change bits or modifying PTE bytes that contain those bits, must be placed after this **ptesync**.

Before permitting an **mtspr** instruction that modifies the LPIDR to be executed on a given thread, software must ensure that no other thread will execute a “conflicting instruction” until after the **mtspr** instruction followed by a context synchronizing instruction have been executed on the given thread (a context synchronizing event can be used instead of the context synchronizing instruction; see Chapter 12).

The “conflicting instructions” in this case are the following.

- A **tlbie** instruction specifying an LPID operand value that matches either the old or the new LPIDR\_LPID value
- A **tlbsync** instruction that is part of a **tlbie/eieio/tlbsync/ptesync** sequence in which the **tlbie** instruction(s) specify an LPID value that matches either the old or the new LPIDR\_LPID value

---

**Programming Note**

The restrictions specified above regarding modifying the LPIDR apply even on systems consisting of only a single-threaded processor, and even if the new LPID value is equal to the old LPID value.

The sequences of operations shown in the following subsections assume a multi-threaded environment. In an environment consisting of only a single-threaded processor, the **tlbsync** must be omitted, and the **eieio** that separates the **tlbie** from the **tlbsync** can be omitted. In a multi-threaded environment, when **tlbie** is used instead of **tlbie** in a Page Table update, the synchronization requirements are the same as when **tlbie** is used in an environment consisting of only a single-threaded processor.

---

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the **ptesync** instruction at the end of the sequence should be followed by a **Store** instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the **ptesync** instruction ensures that if a **Load** instruction executed by another thread returns the chosen value from location X, the sequence’s stores to the Page Table have been performed with respect to that other thread. The **Load** instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE.)

This Note assumes that the Page Table and location X are in storage that is Memory Coherence Required.

---

### 5.10.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The V bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTEA[RPH,LP,AC,R,C,WIMG,PP] ← new values
eieio    /* order 1st update before 2nd */
PTEA[AVA,SM,L,H,V] ← new values (V=1)
ptesync  /* order updates before next Page Table search and before
```

---

---
5.10.1.2 Modifying a Page Table Entry

General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. (The sequence is equivalent to deleting the PTE and then adding a new one; see Sections 5.10.1.1 and 5.10.1.3.)

Modifying the SW field

If the only change being made to a valid entry is to modify the SW field, the following sequence suffices, because the SW field is not used by the hardware and doubleword 0 of the PTE is not modified by the hardware.

```
loop: ldarx r1 ← PTE_dwd_0 /* load dwd 0 of PTE */
      r157:60 ← new SW value /* replace SW in r1 */
      stdcx. PTE_dwd_0 ← r1 /* store dwd 0 of PTE if still reserved (new SW value, other fields unchanged) */
      bne loop /* loop if lost reservation */
```

A ldarx/stbctx., lharc/sthctx., or lwarx/stwctx. pair (specifying the low-order byte, halfword, or word respectively of doubleword 0 of the PTE) can be used instead of the ldarx/stdcx. pair shown above.

Modifying the Virtual Address

If the virtual address translated by a valid PTE is to be modified and the new virtual address hashes to the same PTEG (or the same two PTEGs if the secondary Page Table search is enabled) as does the old virtual address, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTEAVA,SW,L,H,V ← new values (V=1)
ptesync /* order update before tlbie and before next Page Table search */
tlbie(old_B,old_VA14:77-b,old_L,old_LP,old_AP, old_LPID) /*invalidate old translation*/
eieio /* order tlbie before tlbsync */
tlbsync /* order tlbie before ptesync */
ptesync /* order tlbie, tlbsync and 1st update before 2nd update */
```

5.10.1.3 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTEV ← 0 /* (other fields don’t matter) */
ptesync /* order update before tlbie and before next Page Table search */
tlbie(old_B,old_VA14:77-b,old_L,old_LP,old_AP, old_LPID) /*invalidate old translation*/
eieio /* order tlbie before tlbsync */
tlbsync /* order tlbie before ptesync */
ptesync /* order tlbie, tlbsync, and update before next data access */
```
Chapter 6. Interrupts

6.1 Overview

The Power ISA provides an interrupt mechanism to allow the thread to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

6.2 Interrupt Registers

6.2.1 Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Machine Status Save/Restore registers (SRR0 and SRR1). Section 6.5 describes which registers are altered by each interrupt.

Figure 40. Save/Restore Registers

SRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for SRR1 bits in the range 33:36, 42:43, and 45:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set SRR1 (including implementation-dependent setting, e.g. by implementation-specific interrupts).

The SRR0 and SRR1 are hypervisor resources; see Chapter 2.

6.2.2 Hypervisor Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Hypervisor Machine Status Save/Restore registers (HSRR0 and HSRR1). Section 6.5 describes which registers are altered by each interrupt.

Figure 41. Hypervisor Save/Restore Registers

HSRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for HSRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set HSRR1 (including implementation-dependent setting, e.g. by implementation-specific interrupts).

The HSRR0 and HSRR1 are hypervisor resources; see Chapter 2.

6.2.3 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the
interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

Figure 42. Data Address Register

6.2.4 Hypervisor Data Address Register

The Hypervisor Data Address Register (HDAR) is a 64-bit register that is set by the Hypervisor Data Storage Interrupt; see Section 6.5.16. In general, when this interrupt occurs, the HDAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the HDAR set to 0 if the interrupt occurs in 32-bit mode.

Figure 43. Hypervisor Data Address Register

6.2.5 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DSISR is set to indicate the cause of the interrupt.

Figure 44. Data Storage Interrupt Status Register

DSISR bits may be treated as reserved in a given implementation if they are specified as being set either to 0 or to an undefined value for all interrupts that set the DSISR (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts).

6.2.6 Hypervisor Data Storage Interrupt Status Register

The Hypervisor Data Storage Interrupt Status Register (HDSISR) is a 32-bit register that is set by the Hypervisor Data Storage interrupt. In general, when one of these interrupts occurs the HDSISR is set to indicate the cause of the interrupt.

Figure 45. Hypervisor Data Storage Interrupt Status Register

6.2.7 Hypervisor Emulation Instruction Register

The Hypervisor Emulation Instruction Register (HEIR) is a 32-bit register that is set by the Hypervisor Emulation Assistance interrupt; see Section 6.5.18. The image of the instruction that caused the interrupt is loaded into the register.

Figure 46. Hypervisor Emulation Instruction Register

6.2.8 Hypervisor Maintenance Exception Register

Each bit in the Hypervisor Maintenance Exception Register (HMER) is associated with one or more causes of the Hypervisor Maintenance exception, and is set when the associated exception(s) occur. If the corresponding bit in the Hypervisor Maintenance Exception Enable Register (HMEER) is set, a Hypervisor Maintenance Interrupt (HMI) may occur. If the thread is in a power-saving mode when the interrupt would have occurred, the thread will exit the power-saving mode; see Section 6.5.19 and Section 3.3.2.

Figure 47. Hypervisor Maintenance Exception Register

The contents of the HMER are as follows:

- 0 Set to 1 for a Malfunction Alert.
- 1 Set to 1 when performance is degraded for thermal reasons.
- 2 Set to 1 when thread recovery is invoked.
- Others Implementation-specific.

When the mtspr instruction is executed with the HMER as the encoded Special Purpose Register, the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an mthmer instruction.
6.2.9 Hypervisor Maintenance Exception Enable Register

The Hypervisor Maintenance Exception Enable Register (HMEER) is a 64-bit register in which each bit enables the corresponding exception in the HMER to cause the Hypervisor Maintenance interrupt, potentially causing exit from power-saving mode; see Section 6.5.19 and Section 3.3.2.

Figure 48. Hypervisor Maintenance Exception Enable Register

6.2.10 Facility Status and Control Register

The Facility Status and Control Register (FSCR) controls the availability of various facilities in problem state and indicates the cause of a Facility Unavailable interrupt.

When the FSCR makes a facility unavailable, attempted usage of the facility in problem state is treated as follows:

- Execution of an instruction causes a Facility Unavailable interrupt.
- Access of an SPR using mfspr/mtspr causes a Facility Unavailable interrupt
- rfebb, rfid, rfsvc, hrfid and mtmsr[d] instructions have the same effect on bits in system registers as they would if the bits were available.

Programming Note

The FSCR does not prevent rfebb instructions from attempting to set bits in System Registers that the FSCR makes unavailable. Thus changes to BES-CR15 made by the operating system have the potential to result in an illegal transaction state transition when rfebb is subsequently executed in problem state, resulting in the occurrence of a TM Bad Thing type Program interrupt.

The MSR can also make the Transactional Memory facility unavailable in any privilege state, and MMC0_PMCC can make various components of the Performance Monitor unavailable when accessed problem state. An access to one of these facilities when it is unavailable causes a Facility Unavailable interrupt.

When the PCR makes a facility unavailable in problem state, the facility is treated as not implemented in problem state; any Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur.

When a Facility Unavailable interrupt occurs, the unavailable facility that was accessed is indicated in the most-significant byte of the FSCR.

Figure 49. Facility Status and Control Register

The contents of the FSCR are specified below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:7</td>
<td>Interruption Cause (IC)</td>
</tr>
<tr>
<td>02</td>
<td>Access to the DSCR at SPR 3</td>
</tr>
<tr>
<td>03</td>
<td>Access to a Performance Monitor SPR in group A or B when MMCR0_PMCC is set to a value for which the access results in a Facility Unavailable interrupt (See the definition of MMCR0_PMCC in Section 9.4.4.)</td>
</tr>
<tr>
<td>04</td>
<td>Execution of a BHRB Instruction</td>
</tr>
<tr>
<td>05</td>
<td>Access to a Transactional Memory SPR or execution of a Transactional Memory Instruction</td>
</tr>
<tr>
<td>06</td>
<td>Reserved</td>
</tr>
<tr>
<td>07</td>
<td>Access to an Event-Based Branch SPR or execution of an Event-Based Branch Instruction</td>
</tr>
<tr>
<td>08</td>
<td>Access to the Target Address Register</td>
</tr>
<tr>
<td>08:63</td>
<td>Facility Enable (FE)</td>
</tr>
<tr>
<td></td>
<td>The FE field controls the availability of various facilities in problem state as specified below.</td>
</tr>
<tr>
<td>8:54</td>
<td>Reserved</td>
</tr>
<tr>
<td>55</td>
<td>Target Address Register (TAR)</td>
</tr>
<tr>
<td>0</td>
<td>The TAR and bctar instruction are not available in problem state.</td>
</tr>
<tr>
<td>1</td>
<td>The TAR and bctar instruction are available in problem state unless made unavailable by another register.</td>
</tr>
<tr>
<td>56</td>
<td>Event-Based Branch Facility (EBB)</td>
</tr>
</tbody>
</table>
The Event-Based Branch facility SPRs and instructions are not available in problem state, and event-based exceptions and branches do not occur.

The Event-Based Branch facility SPRs and instructions are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

The Event-Based Branch facility SPRs and instructions are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

HFSCR58:60 are used to control the availability of Transactional Memory, the Performance Monitor, and the BHRB in problem and privileged states. FSCR58:60 are reserved since the availability of Transactional Memory is controlled by the MSR, and the availability of the Performance Monitor and BHRB is controlled by MMCR0.

When an OS has set the FSCR such that a facility is unavailable, the OS should either emulate the facility when it is accessed or provide an application interface that requires the application to request use of the facility before it accesses the facility.

When the PCR makes a facility unavailable in problem state, the facility is treated as not implemented in problem state; any Hypervisor Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur as a result of problem-state access. See Section 2.6 for additional information.)

When a Hypervisor Facility Unavailable interrupt occurs, the facility that was accessed is indicated in the most-significant byte of the HFSCR.

When the PCR makes a facility unavailable in problem state, the facility is treated as not implemented in problem state; any Hypervisor Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur as a result of problem-state access. See Section 2.6 for additional information.)

When a Hypervisor Facility Unavailable interrupt occurs, the facility that was accessed is indicated in the most-significant byte of the HFSCR.

<table>
<thead>
<tr>
<th>IC</th>
<th>Facility Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 50. Hypervisor Facility Status and Control Register

The contents of the HFSCR are specified below.

Value | Meaning
--- | ---
0:7 | **Interruption Cause (IC)**

When a Hypervisor Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the access that was attempted. The values and their meanings are specified below.

- **00** Access to a Floating Point register or execution of a Floating Point instruction
- **01** Access to a Vector or VSX register or execution of a Vector or VSX instruction
- **02** Access to the DSCR at SPRs 3 or 17
- **03** Read or write access of a Performance Monitor SPR in group A, or read access of a Performance Monitor SPR in group B

0 The Event-Based Branch facility SPRs and instructions are not available in problem state, and event-based exceptions and branches do not occur.

1 The Event-Based Branch facility SPRs and instructions are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

57 Reserved

58:60 Reserved

57 Reserved

58:60 Reserved

61 Data Stream Control Register at SPR 3 (DSCR)

0 SPR 3 is not available in problem state.

1 SPR 3 is available in problem state unless made unavailable by another register.

62:63 Reserved

6.2.11 Hypervisor Facility Status and Control Register

The Hypervisor Facility Status and Control Register (HFSCR) controls the available of various facilities in problem and privileged states, and indicates the cause of a Hypervisor Facility Unavailable interrupt.

When the HFSCR makes a facility unavailable, attempted usage of the facility in problem or privileged states is treated as follows:

- Execution of an instruction causes a Hypervisor Facility Unavailable interrupt.
- Access of an SPR using mfspr/mtspr causes a Hypervisor Facility Unavailable interrupt
- rfebb, rfid, rfscv, [h]rfid and mtmsr[d] instructions have the same effect on bits in system registers as they would if the bits were available.

Because the HFSCR does not prevent mtspr, rfscv, [h]rfid, and mtmsr[d] instructions from setting bits in system registers that the HFSCR will make unavailable after a transition to a lower privilege state, these instructions may cause interrupts in a variety of unexpected ways. For example, consider a hypervisor that sets HSRR1 such that hrfid returns to a lower privilege state with MSR[TS] nonzero. A TM Bad Thing interrupt will result, despite that TM is made unavailable by the HFSCR.

Similarly, the HFSCR does not prevent rfebb instructions from attempting to set bits in System Registers that the HFSCR makes unavailable. Thus changes to BESCR[TS] made by the hypervisor have the potential to result in an illegal transaction state transition when rfebb is subsequently executed in problem or privileged state, resulting in the occurrence of a TM Bad Thing type Program interrupt.

When the PCR makes a facility unavailable in problem state, the facility is treated as not implemented in problem state; any Hypervisor Facility Unavailable interrupt that would occur if the facility were not made unavailable by the PCR does not occur as a result of problem-state access. See Section 2.6 for additional information.)

When a Hypervisor Facility Unavailable interrupt occurs, the facility that was accessed is indicated in the most-significant byte of the HFSCR.
| 0 | Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are not available in problem and privileged states; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are not available in privileged state. (See the definition of MMCR0PMCC in Section 9.4.4 for a definition of groups A and B.) |
| 1 | Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are available in problem and privileged states unless made unavailable by another register; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are available in privileged state. |
| 8:63 | **Facility Enable (FE)** |
| The FE field controls the availability of various facilities in problem and privileged states as specified below. |
| 8:54 | Reserved |
| 55 | **Target Address Register (TAR)** |
| 0 | The TAR and `bctar` instruction are not available in problem and privileged state. |
| 1 | The TAR and `bctar` instruction are available in problem and privileged state unless made unavailable by another register. |
| 56 | **Event-Based Branch Facility (EBB)** |
| 0 | The Event-Based Branch facility SPRs and instructions are not available in problem and privileged states, and event-based exceptions and branches do not occur. |
| 1 | The Event-Based Branch facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other bits. |
| 58 | **Transactional Memory Facility (TM)** |
| 0 | The Transactional Memory Facility SPRs and instructions are not available in problem and privileged states. |
| 1 | The Transactional Memory Facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register. |
| 59 | **BHRB Instructions (BHRB)** |
| 0 | The BHRB instructions (`mfbhrb`, `clrbhrb`) are not available in problem and privileged states. |
| 1 | The BHRB instructions (`mfbhrb`, `clrbhrb`) are available in problem and privileged states unless made unavailable by another register. |
| 60 | **Performance Monitor Facility SPRs (PM)** |
| 0 | Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are not available in problem and privileged states; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are not available in privileged state. (See the definition of MMCR0PMCC in Section 9.4.4 for a definition of groups A and B.) |
| 1 | Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are available in problem and privileged states unless made unavailable by another register; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are available in privileged state. |
| 61 | **Data Stream Control Register (DSCR)** |
| 0 | SPR 3 is not available in problem or privileged states and SPR 17 is not available in privileged state. |
| 1 | SPR 3 is available in problem and privileged states and SPR 17 is available in privileged state unless made unavailable by another register. |
| 62 | **Vector and VSX Facilities (VECVSX)** |
| 0 | The facilities whose availability is controlled by either MSR_VEC or MSR_VSX are not available in problem and privileged states. |
| 1 | The facilities whose availability is controlled by either MSR_VEC or MSR_VSX are available in problem and privileged states unless made unavailable by another register. |
| 63 | **Floating Point Facility (FP)** |
| 0 | The facilities whose availability is controlled by MSR_FP are not available in problem and privileged states. |
| 1 | The facilities whose availability is controlled by MSR_FP are available in problem and privileged states unless made unavailable by another register. |
The FSCR can be used to determine whether a particular facility is being used by an application, and the HFSCR can be used to determine whether a particular facility is being used by either an application or by an operating system. This is done by disabling the facility initially, and enabling it in the interrupt handler upon first usage. The information about the usage of a particular facility can be used to determine whether that facility's state must be saved and restored when changing program context.
### Programming Note

The following tables summarize the interrupts that occur as a result of accessing the non-privileged Performance Monitor registers in problem state when MMCR0_PMC0, PCR, and HFSCR are set to various values. (Accesses to privileged Performance Monitor SPRs (SPRs 784-792, 795-798) in problem state result in Privileged Instruction Type Program interrupts.)

<table>
<thead>
<tr>
<th>SPR</th>
<th>#</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMCR2</td>
<td>769</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>MMCR0</td>
<td>779</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>MMCRA</td>
<td>770</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC1</td>
<td>771</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC2</td>
<td>772</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC3</td>
<td>773</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC4</td>
<td>774</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC5</td>
<td>775</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>PMC6</td>
<td>776</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
<tr>
<td>MMCR0</td>
<td>779</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
<td>HU</td>
<td>HE</td>
<td>HU</td>
<td>FU</td>
<td>HU</td>
</tr>
</tbody>
</table>

#### Notes:

4. Terminology:
   - FU: Facility Unavailable interrupt
   - HE: Hypervisor Emulation Assistance interrupt
   - HU: Hypervisor Facility Unavailable interrupt

5. Write operations to this SPR are not implemented for all privilege states. (See the `mfspr` instruction description in Section 4.4.4 for additional information.) FU or HU interrupts do not occur regardless of the value of MMCR0_PMC0 or HFSCR.

6. When the PCR indicates a version of the architecture prior to V 2.07, this SPR is treated as not implemented in problem state; no FU or HU interrupts occur regardless of the value of MMCR0_PMC0 or HFSCR.

7. An HU interrupt occurs if HFSCR=0 when this register is accessed in either problem state or privileged (but not hypervisor) state.
6.3 Interrupt Synchronization

When an interrupt occurs, SRR0 or HSRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0 or HSRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.5.1. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.5.1, but does satisfy items 4 and 5.

6.4 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are “system-caused” are:

- System Reset
- Machine Check
- External
- Decrementer
- Directed Privileged Doorbell
- Hypervisor Decrementer
- Hypervisor Maintenance
- Directed Hypervisor Doorbell
- Performance Monitor

External, Decrementer, Hypervisor Decrementer, Directed Privileged Doorbell, Directed Hypervisor Doorbell, and Hypervisor Maintenance interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

“Instruction-caused” interrupts are further divided into two classes, precise and imprecise.

6.4.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or the immediately following instruction.

Which instruction is addressed can be determined from the interrupt type and status bits.

2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread.

3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.

4. Architecturally, no subsequent instruction has begun execution.

6.4.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 6.5.9, “Program Interrupt” on page 959.

2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing thread.

3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 6.5.9.

4. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.
6.4.3 Interrupt Processing

Associated with each kind of interrupt is an interrupt vector, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the thread’s state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 6.5.2) differs from the description given below in several respects.

1. SRR0 or HSRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.

2. Bits 33:36 and 42:47 of SRR1 or HSRR1 are loaded with information specific to the interrupt type.

3. Bits 0:32, 37:41, and 48:63 of SRR1 or HSRR1 are loaded with a copy of the corresponding bits of the MSR.

4. The MSR is set as shown in Figure 51 on page 951. In particular, MSR bits IR and DR are set as specified by LPCRAIL (see Section 2.2), and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect beginning with the first instruction executed following the interrupt.

5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 52 on page 952. An offset may be applied to get the effective addresses, as specified by LPCRAIL (see Section 2.2).

Interrupts do not clear reservations obtained with lbarx, lharx, lwarx, ldarx, or lqarx.

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the operating system before dispatching a “new” program.

- **stbcx, sthcx, stwcx, stdcx, or stqcx**, to clear the reservation if one is outstanding, to ensure that a lbarx, lharx, lwarx, ldarx, or lqarx in the interrupted program is not paired with a stbcx, sthcx, stwcx, stdcx, or stqcx on the “new” program.

- **sync**, to ensure that all storage accesses caused by the interrupted program will be performed with respect to another thread before the program is resumed on that other thread.

- **isync or rfid**, to ensure that the instructions in the “new” program execute in the “new” context.

- **treclaim**, to ensure that any previous use of the transactional facility is terminated.
For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:

- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by `lmw` for which the storage operand is not aligned, or by `dcbz` for which the storage operand is in storage that is Write Through Required or Caching Inhibited.

- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: A Hypervisor Emulation Assistance interrupt is caused by `imw` for which the instruction description states, potentially to cause the interrupt. Example: Alignment interrupt caused by `lwax` for which the storage operand is not aligned.

- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should be permitted to access but for which there currently is no PTE that satisfies the Page Table search.)

In general, the instruction should not be emulated if:

- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by `sc`.

- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by `lwax` for which the storage operand is not aligned.

- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should be permitted to access but for which there currently is no PTE that satisfies the Page Table search.)
6.4.4 Implicit alteration of HSRR0 and HSRR1

Executing some of the more complex instructions may have the side effect of altering the contents of HSRR0 and HSRR1. The instructions listed below are guaranteed not to have this side effect. Any omission of instruction suffixes is significant; e.g., add is listed but add. is excluded.

1. Branch instructions
   - bl[a], bc[l][a], bcr[l], bctcr[l]
2. Fixed-Point Load and Store Instructions
   - lbz, lbzx, lh, lhz, lw, lwz, ld<64>, ld<64>,
   - stb, stbx, sth, sthx, stw, stwx, std<64>,
   - stdx<64>

Execution of these instructions is guaranteed not to have the side effect of altering HSRR0 and HSRR1 only if the storage operand is aligned and MSRDR=0.

3. Arithmetic instructions
   - addi, adds, add, subf, neg

4. Compare instructions
   - cmpi, cmp, cmpli, cmpl

5. Logical and Extend Sign instructions
   - ori, oris, xor, xoris, and, or, xor, and, or, eqv, andc, orc, extsb, extsh, extsw

6. Rotate and Shift instructions
   - rldicl<64>, rldicr<64>, rldic<64>, rlwinm,
   - rldcl<64>, rldcr<64>, rlwinm, rldimi<64>, rlwimi,

7. Other instructions
   - isync
   - rfid, hrfid
   - mtspr, mfspr, mtmsrd, mfmsr

These instructions, and the other excluded instructions, may be implemented with the assistance of the Hypervisor Emulation Assistance interrupt, or of implementation-specific interrupts that modify HSRR0 and HSRR1. The included instructions are guaranteed not to be implemented thus. (The included instructions are sufficiently simple as to be unlikely to need such assistance. Moreover, they are likely to be needed in interrupt handlers before HSRR0 and HSRR1 have been saved or after HSRR0 and HSRR1 have been restored.)

Similarly, fetching instructions may have the side effect of altering the contents of HSRR0 and HSRR1 unless MSRIR=0.

Programming Note

In order to handle Machine Check and System Reset interrupts correctly, the operating system should manage MSRRI as follows.

- In the Machine Check and System Reset interrupt handlers, interpret SRR1 bit 62 (where MSRRI is placed) as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable

- In each interrupt handler, when enough state has been saved that a Machine Check or System Reset interrupt can be recovered from, set MSRRI to 1.

- In each interrupt handler, do the following (in order) just before returning.
  1. Set MSRRI to 0.
  2. Set SRR0 and SRR1 to the values to be used by rfid. The new value of SRR1 should have bit 62 set to 1 (which will happen naturally if SRR1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  3. Execute rfid.

For interrupts that set the SRRs other than Machine Check or System Reset, MSRRI can be managed similarly when these interrupts occur within interrupt handlers for other interrupts that set the SRRs.

This Note does not apply to interrupts that set the HSRRs because these interrupts put the thread into hypervisor state, and either do not occur or can be prevented from occurring within interrupt handlers for other interrupts that set the HSRRs.
6.5 Interrupt Definitions

Figure 51 shows all the types of interrupts and the values assigned to the MSR for each. Figure 52 shows the effective address of the interrupt vector for each interrupt type. (Section 5.7.4 on page 895 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 52.)

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>MSR Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IR DR FE0 FE1 EE RI ME HV</td>
</tr>
<tr>
<td>System Reset</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>Machine Check</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>Data Storage</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Data Segment</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Instruction Storage</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Instruction Segment</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>External</td>
<td>r r 0 0 0 h - e</td>
</tr>
<tr>
<td>Alignment</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Program</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>FP Unavailable</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Decrementer</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Directed Privileged Doorbell Interrupt</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Hypervisor Decrementer</td>
<td>r r 0 0 0 0 - - 1</td>
</tr>
<tr>
<td>System Call</td>
<td>r r 0 0 0 0 0 - s</td>
</tr>
<tr>
<td>Trace</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Hypervisor Data Storage</td>
<td>r r 0 0 0 0 - - 1</td>
</tr>
<tr>
<td>Hypervisor Instr. Storage.</td>
<td>0 0 0 0 0 0 - - 1</td>
</tr>
<tr>
<td>Hypv Emulation Assistance</td>
<td>r r 0 0 0 0 - - 1</td>
</tr>
<tr>
<td>Hypervisor Maintenance</td>
<td>0 0 0 0 0 0 - - 1</td>
</tr>
<tr>
<td>Directed Hypervisor Doorbell Interrupt</td>
<td>r r 0 0 0 0 - - 1</td>
</tr>
<tr>
<td>Performance Monitor</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Vector Unavailable</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>VSX Unavailable</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Facility Unavailable</td>
<td>r r 0 0 0 0 - -</td>
</tr>
<tr>
<td>Hypervisor Facility Unavailable</td>
<td>r r 0 0 0 0 - - 1</td>
</tr>
</tbody>
</table>
Figure 51. MSR setting due to interrupt

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>MSR Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IR DR FE0 FE1 EE RI ME HV</td>
</tr>
<tr>
<td>0</td>
<td>bit is set to 0</td>
</tr>
<tr>
<td>1</td>
<td>bit is set to 1</td>
</tr>
<tr>
<td>-</td>
<td>bit is not altered</td>
</tr>
<tr>
<td>r</td>
<td>if LPCRайл = 2 or 3, MSRIR = 0b11, and the interrupt does not cause a transition from MSRHV = 0 to MSRHV = 1, set to 1; otherwise set to 0</td>
</tr>
<tr>
<td>p</td>
<td>if the interrupt occurred while the thread was in power-saving mode, set to 1; otherwise not altered</td>
</tr>
<tr>
<td>e</td>
<td>if LPES = 0, set to 1; otherwise not altered</td>
</tr>
<tr>
<td>h</td>
<td>if LPES = 1, set to 0; otherwise not altered</td>
</tr>
<tr>
<td>s</td>
<td>if LEV = 1, set to 1; otherwise not altered</td>
</tr>
</tbody>
</table>

Settings for Other Bits:

Bits BE, FP, PR, SE, TM\(^5\), SLE, VEC\(^1\), VSX\(^2\), and PMM are set to 0.

If the interrupt results in HV being equal to 1, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the LPCRILE bit.

The SF bit is set to 1.

If the TS field contained 0b10 (Transactional) when the interrupt occurred, the TS field is set to 0b01 (Suspended); otherwise the TS field is not altered.

Reserved bits are set as if written as 0.

1 Category: Vector
2 Category: Vector Scalar Emulation
3 Category: Floating-Point
4 Category: Server, Processor Control
5 Category: Transactional Memory

Figure 51. MSR setting due to interrupt
## 6.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a Machine Check exception or a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

When the thread is in any power-saving level, a System Reset interrupt occurs when a System Reset exception exists. When the thread is in doze or nap power-saving levels, a System Reset interrupt occurs when any of the following exceptions exists provided that the exception is enabled to cause exit from power saving mode (see Section 2.2, "Logical Partitioning Control Register (LPCR)"). When the thread is in sleep or rvwinkle power-saving level, it is implementation-specific whether the following exceptions, when enabled, cause exit, or whether only a system-reset causes exit.

- **External**
- **Decrementer**
- **Directed Privileged Doorbell <S.PC>**
- **Directed Hypervisor Doorbell <S.PC>**
- **Hypervisor Maintenance**
- **Implementation-specific**

SRR1 indicates the exception that caused exit from power-saving mode as specified below.

The following registers are set:

- **SRR0**  
  - If the interrupt did not occur when the thread was in power-saving mode, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present; otherwise, set to an undefined value.

- **SRR1**  
  - **33**  
    - Implementation-dependent.

- **34:36**  
  - Set to 0.

- **42:45**  
  - If the interrupt did not occur when the thread was in power-saving mode, set to an implementation-specific value. If the interrupt occurred when the thread was in power-saving mode, set to indicate the

---

**Programming Note**

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 52 risks incompatibility with future implementations.

---

### Table: Effective Address of Interrupt Vector by Interrupt Type

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Interrupt Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00..0000_0100</td>
<td>System Reset</td>
</tr>
<tr>
<td>00..0000_0200</td>
<td>Machine Check</td>
</tr>
<tr>
<td>00..0000_0300</td>
<td>Data Storage</td>
</tr>
<tr>
<td>00..0000_0380</td>
<td>Data Segment</td>
</tr>
<tr>
<td>00..0000_0400</td>
<td>Instruction Storage</td>
</tr>
<tr>
<td>00..0000_0480</td>
<td>Instruction Segment</td>
</tr>
<tr>
<td>00..0000_0500</td>
<td>External</td>
</tr>
<tr>
<td>00..0000_0600</td>
<td>Alignment</td>
</tr>
<tr>
<td>00..0000_0700</td>
<td>Program</td>
</tr>
<tr>
<td>00..0000_0800</td>
<td>Floating-Point Unavailable</td>
</tr>
<tr>
<td>00..0000_0900</td>
<td>Decrementer</td>
</tr>
<tr>
<td>00..0000_0980</td>
<td>Hypervisor Decrementer</td>
</tr>
<tr>
<td>00..0000_0A00</td>
<td>Directed Privileged Doorbell</td>
</tr>
<tr>
<td>00..0000_0B00</td>
<td>Reserved</td>
</tr>
<tr>
<td>00..0000_0C00</td>
<td>System Call</td>
</tr>
<tr>
<td>00..0000_0D00</td>
<td>Trace</td>
</tr>
<tr>
<td>00..0000_0E00</td>
<td>Hypervisor Data Storage</td>
</tr>
<tr>
<td>00..0000_0E20</td>
<td>Hypervisor Instruction Storage</td>
</tr>
<tr>
<td>00..0000_0E40</td>
<td>Hypervisor Emulation Assistance</td>
</tr>
<tr>
<td>00..0000_0E60</td>
<td>Hypervisor Maintenance</td>
</tr>
<tr>
<td>00..0000_0E80</td>
<td>Directed Hypervisor Doorbell</td>
</tr>
<tr>
<td>00..0000_0E90</td>
<td>Reserved</td>
</tr>
<tr>
<td>00..0000_0E90</td>
<td>Reserved for implementation-dependent interrupt for performance monitoring</td>
</tr>
<tr>
<td>00..0000_0F00</td>
<td>Performance Monitor</td>
</tr>
<tr>
<td>00..0000_0F20</td>
<td>Vector Unavailable</td>
</tr>
<tr>
<td>00..0000_0F40</td>
<td>VSX Unavailable</td>
</tr>
<tr>
<td>00..0000_0F40</td>
<td>Facility Unavailable</td>
</tr>
<tr>
<td>00..0000_0F80</td>
<td>Hypervisor Facility Unavailable</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>00..0000_0FFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

1. The values in the Effective Address column are interpreted as follows.
   - 00...0000_0nnn means 0x0000_0000_0000_0nnn unless the values of LPCRAIL and MSRHVIRDR cause the application of an effective address offset. See the description of LPCRAIL in Section 2.2 for more details.

2. Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.

4. Category: Vector Scalar Extension
5. Category: Floating Point
6. Category: Server Processor Control

---

**Figure 52. Effective address of interrupt vector by interrupt type**

The exception that caused exit from power-saving mode.
ing mode as shown below:

<table>
<thead>
<tr>
<th>SRR142:45</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0001</td>
<td>Reserved</td>
</tr>
<tr>
<td>0010</td>
<td>Implementation specific</td>
</tr>
<tr>
<td>0011</td>
<td>Directed Hypvsr Doorbell</td>
</tr>
<tr>
<td>0100</td>
<td>System Reset</td>
</tr>
<tr>
<td>0101</td>
<td>Directed Privlgd Doorbell</td>
</tr>
<tr>
<td>0110</td>
<td>Decrementer</td>
</tr>
<tr>
<td>0111</td>
<td>Reserved</td>
</tr>
<tr>
<td>1000</td>
<td>External</td>
</tr>
<tr>
<td>1001</td>
<td>Reserved</td>
</tr>
<tr>
<td>1010</td>
<td>Hypervisor Maintenance</td>
</tr>
<tr>
<td>1011</td>
<td>Reserved</td>
</tr>
<tr>
<td>1100</td>
<td>Implementation specific</td>
</tr>
<tr>
<td>1101</td>
<td>Reserved</td>
</tr>
<tr>
<td>1110</td>
<td>Implementation specific</td>
</tr>
<tr>
<td>1111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

If multiple exceptions that cause exit from power-saving mode exist, the exception reported is the exception corresponding to the interrupt that would have occurred if the same exceptions existed and the thread was not in power-saving mode.

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows:

| 00 | The interrupt did not occur when the thread was in power-saving mode. |
| 01 | The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode. |
| 10 | The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. |

11 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

**Programming Note**

Although the resources that are maintained in power-saving mode (except in doze power-saving level) are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of SRR146:47, to determine what state to restore. (To avoid implementation-dependence in the portion of the hypervisor that enters power-saving mode, the hypervisor must use the specification of the four instructions to determine what state to save.)

| 62 | If the interrupt did not occur while the thread was in power-saving mode, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in power-saving mode, set to 1 if the thread is in a recoverable state; otherwise set to 0. |
| Others | Loaded from the MSR. |

### MSR

See Figure 51 on page 951.

In addition, if the interrupt occurs when the thread is in power-saving mode and is caused by an exception other than a System Reset exception, all other registers, except HSRR0 and HSRR1, that would be set by the corresponding interrupt if the exception occurred when the thread was not in power-saving mode are set by the System Reset interrupt, and are set to the values to which they would be set if the exception occurred when the thread was not in power-saving mode.

Execution resumes at effective address 0x0000_0000_0000_0100.

The means for software to distinguish between power-on Reset and other types of System Reset are implementation-dependent.

### 6.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check
interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 5.6), or by an error in the storage subsystem.

When the thread is not in power-saving mode, Machine Check interrupts are enabled when MSR<sub>ME</sub>=1; if MSR<sub>ME</sub>=0 and a Machine Check exception occurs, the thread enters the Checkstop state. When the thread is in doze or nap power-saving levels, Machine Check interrupts are treated as enabled when LPCR<sub>PECE</sub>[2]=1 and cannot occur when LPCR<sub>PECE</sub>[3]=0. When the thread is in sleep or rrwinkle power-saving level, it is implementation-specific whether Machine Check interrupts are treated as enabled under the same conditions as in doze and nap power-saving level or if they cannot occur. If a Machine Check exception occurs while the thread is in power-saving mode and the Machine Check exception is not enabled to cause exit from power-saving mode, the result is implementation specific.

The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 5.6), or if an implementation-dependant hardware error occurs that prevents continued operation.

### Disabled Machine Check (Checkstop State)

When a thread is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the thread. Some implementations may preserve some or all of the internal state of the thread when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

### Enabled Machine Check

If a Machine Check exception causes an interrupt that is not context synchronizing or causes the loss of a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

In some systems, the operating system may attempt to identify and log the cause of the Machine Check.

The following registers are set:

#### SRR0

- If the interrupt did not occur while the thread was in power-saving mode, set on a "best effort" basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred; otherwise set to an undefined value.

#### SRR1

- 00 The interrupt did not occur when the thread was in power-saving mode.
- 01 The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.
- 10 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution.
- 11 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

### Programming Note

Since the hypervisor can save the address of the instruction following the Power-Saving Mode instruction if needed, there is no need for the thread to preserve it and store it into SRR0. Therefore, for ease of implementation, the contents of SRR0 upon exit from power-saving mode are specified to be undefined.
A Machine Check interrupt caused by the existence of multiple SLB entries or TLB entries (or similar entries in implementation-specific translation caches) which translate a given effective or virtual address (see Sections 5.7.6.2 and 5.7.7.3.) must occur while still in the power-saving mode, set to 1 if the thread is in a recoverable state; otherwise set to 0.

The access violates Basic Storage Protection.

The access violates Virtual Page Class Key Storage Protection and LPCRKBV=0.

A Data Address Watchpoint match occurs.

Execution of an ecowx or eciwx instruction is disallowed because EARn=0.

An attempt is made to execute a Fixed-Point Load or Store Caching Inhibited instruction with MSRDR=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

If a stbcx, sthcx, stwcx, stdcx, or stdcx, instruction would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional Store to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed instruction, a Data Storage interrupt does not occur.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: Set to 0.
- **MSR**: Loaded from the MSR.
- **DSISR**: See Figure 51.
- **DAR**: Set to an implementation-dependent value.
- **Others**: Set to 1 if MSRDR=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.
- **Others**: Set to 0.

If a **lq**, **stq**, **lbarx**, **lbarwx**, **ldarx**, **ldarwx**, **lqarx**, **stbcx**, **stbcwx**, **stwcx**, **stwcwx**, **stdcx**, or **stqcx** instruction refers to storage that is Write Through Required or Caching Inhibited.

### Programming Note

Although the resources that are maintained in power-saving mode (except in the doze power-saving level) are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of SRR146:47, to determine what state to restore. (To avoid implementation-dependence in the portion of the hypervisor that enters power-saving mode, the hypervisor must use the specification of the four instructions to determine what state to save.)

### Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, which may be placed into registers. This corruption of register contents may occur even if the interrupt is recoverable.
tion that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.

38: Set to 1 for a Store, dcbz, or ecowx instruction; otherwise set to 0.

39: Set to 0.

40: Set to 1 for a Store, dcbz, or ecowx instruction; otherwise set to 0.

41: Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

42: Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43: Set to 1 if execution of an ecixw or ecowx instruction is attempted when EAR_E=0; otherwise set to 0.

44:61: Set to 0.

62: Set to 1 if an attempt is made to execute a Fixed-Point Load or Store Caching Inhibited instruction with MSR_DR=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

63: Set to 0.

DAR

Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a Load Word instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the DSISR), the DAR is set to the effective address of a byte in the first aligned double-word for which access was attempted in the page that caused the exception.

- a Data Storage exception occurs for reasons other than a Data Address Watchpoint match or, for ecixw and ecowx, EAR_E=0
  - a byte in the block that caused the exception, for a Cache Management instruction
  - a byte in the first aligned quadword for which access was attempted in the page that caused the exception, for a quadword Load or Store instruction (i.e., a Load or Store instruction for which the storage operand is a quadword; “first” refers to address order; see Section 6.7)
  - a byte in the first aligned double-word for which access was attempted in the page that caused the exception, for a non-quadword Load or Store instruction or an ecixw or ecowx instruction

- undefined, for a Data Address Watchpoint match, or if ecixw or ecowx is executed when EAR_E=0

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR. However, if one or more DSI-causing exceptions occur together with a Virtualized Page Class Key Storage Protection exception that occurs when LPCR_KBV=1 and Virtualized Partition Memory is disabled by VPM1=0, an HDSI results, and all of the exceptions are reported in the HDSISR.

Execution resumes at effective address 0x0000_0000_0000_0000, possibly offset as specified in Figure 52.

6.5.4 Data Segment Interrupt

A Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a Load, Store, icbi, dcbz, dcbst, dcbf[l], ecixw, or ecowx instruction cannot be translated to a virtual address.

If a stbcx., sthcx., stwcx., stdcx., or stqcx. would not perform its store in the absence of a Data Segment interrupt and a non-conditional Store to the specified effective address would cause a Data Segment interrupt, it is implementation-dependent whether a Data Segment interrupt occurs.

If the XER specifies a length of zero for an indexed Move Assist instruction, a Data Segment interrupt does not occur.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1 33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

DSISR Set to an undefined value.

DAR Set to the effective address of a storage element as described in the following list.

- a byte in the block that caused the exception, for a Cache Management instruction
- a byte in the first aligned quadword for which access was attempted in the
segment that caused the exception, for
a quadword Load or Store instruction
(i.e., a Load or Store instruction for
which the storage operand is a quad-
word; “first” refers to address order:
see Section 6.7)
- a byte in the first aligned doubleword
for which access was attempted in the
segment that caused the exception, for
a non-quadword Load or Store instruc-
tion or an ecxwx or ecowx instruction

If the interrupt occurs in 32-bit mode the
high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address
0x0000_0000_0000_0380, possibly offset as specified
in Figure 52.

--- Programming Note ---
A Data Segment interrupt occurs if MSR_DR=1 and
the translation of the effective address of any byte
of the specified storage location is not found in the
SLB (or in any implementation-specific address
translation lookaside information).

6.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher
priority exception exists, the value of the expression
\[(MSR_{HV PR} = 0b10) \mid (\neg VPM_0 \& \neg MSR_{IR})
\mid (\neg VPM_1 \& MSR_{IR})\]
is 1, and the next instruction to be executed cannot be
fetched for any of the following reasons.

- Instruction address translation is enabled and the
  virtual address cannot be translated to a real
  address.
- The fetch access violates storage protection.

The following registers are set:

**SRR0**
- Set to the effective address of the instruction
  that the thread would have attempted to exe-
  cute next if no interrupt conditions were
  present (if the interrupt occurs on attempting
  to fetch a branch target, SRR0 is set to the
  branch target address).

**SRR1**
- 33: Set to 1 if MSR_{IR}=1 and the translation for
  an attempted access is not found in the
  Page Table; otherwise set to 0.
- 34: Set to 0.
- 35: Set to 1 if the access is to No-execute or
  Guarded storage; otherwise set to 0.
- 36: Set to 1 if the access is not permitted by
  Figure 32 or 33, as appropriate; otherwise
  set to 0.
- 42: Set to 1 if the access is not permitted by vir-
  tual page class key protection; otherwise
  set to 0.
- 43: Set to 0.

**MSR**
- Loaded from the MSR.

Execution resumes at effective address
0x0000_0000_0000_0400, possibly offset as specified
in Figure 52.

--- Programming Note ---
An Instruction Segment interrupt occurs if
MSR_{IR}=1 and the translation of the effective
address of the next instruction to be executed is not
found in the SLB (or in any implementation-specific
address translation lookaside information).

6.5.6 Instruction Segment
Interrupt

An Instruction Segment interrupt occurs when no
higher priority exception exists and the next instruction
to be executed cannot be fetched because instruction
address translation is enabled and the effective
address cannot be translated to a virtual address.

The following registers are set:

**SRR0**
- Set to the effective address of the instruction
  that the thread would have attempted to exe-
  cute next if no interrupt conditions were
  present (if the interrupt occurs on attempting
  to fetch a branch target, SRR0 is set to the
  branch target address).

**SRR1**
- 33: Set to 0.
- 42: Set to 0.

**Others**
- Loaded from the MSR.

**MSR**
- See Figure 51 on page 951.

Execution resumes at effective address
0x0000_0000_0000_0480, possibly offset as specified
in Figure 52.

--- Programming Note ---
An Instruction Segment interrupt occurs if
MSR_{IR}=1 and the translation of the effective
address of the next instruction to be executed is not
found in the SLB (or in any implementation-specific
address translation lookaside information).

6.5.7 External Interrupt

An External interrupt is classified as being either a
Direct External interrupt or a Mediated External inter-
rupt. Throughout this Book, usage of the phrase “Exter-
nal interrupt”, without further classification, refers to
both a Direct External interrupt and a Mediated External interrupt.

### 6.5.7.1 Direct External Interrupt

A Direct External interrupt occurs when no higher priority exception exists, a Direct External exception exists, and the value of the expression

$$\text{MSREE} \land (\neg\text{LPES}_0 \land (\neg\text{MSRH}_V \land \text{MSRP}))$$

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When LPES$0$=0, the following registers are set:

- **HSRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **Others**: Loaded from the MSR.

When LPES$0$=1, the following registers are set:

- **SRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **Others**: Loaded from the MSR.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 52.

### Programming Note

Because the value of MSREE is always 1 when the thread is in problem state, the simpler expression

$$\text{MSREE} \land (\neg\text{LPES}_0 \lor \neg\text{MSRH}_V)$$

is equivalent to the expression given above.

### Programming Note

The Direct External exception has the same meaning as the External exception in versions of the architecture prior to Version 2.05.

### 6.5.7.2 Mediated External Interrupt

A Mediated External interrupt occurs when no higher priority exception exists, a Mediated External exception exists (see the definition of LPCRMER in Section 2.2), and the value of the expression

$$\text{MSREE} \land (\neg\text{MSRH}_V \lor \text{MSRP})$$

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When LPES$0$=0, the following registers are set:

- **HSRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **Others**: Loaded from the MSR.

When LPES$0$=1, the following registers are set:

- **SRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **Others**: Loaded from the MSR.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 52.

### 6.5.8 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The operand of a Floating-Point Load, a Floating-Point Store, a Vector-Scalar Extension Load, or a Vector-Scalar Extension Store is not word-aligned, or crosses a virtual page boundary.
- The operand of lq, stq, lmw, stmw, lharx, lwarx, ldarx, stcx, stwcx, stdcx, eciwx, ecowx, itdp, ifdp, stfdp, or stfdpx is not aligned.
- The operand of a Load or Store is not aligned and the thread is in Little-Endian mode.
- The instruction is lmw, stmw, lswi, lswx, stswi, or stswx, and the operand is in storage that is Write Through Required or Caching Inhibited, or the thread is in Little-Endian mode.
- The operand of a Load or Store crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.
- The operand of a Load or Store is not aligned and is in storage that is Write Through Required or Caching Inhibited.
The operand of \texttt{lfdp}, \texttt{lfdp}, \texttt{stfdp}, \texttt{stfdpx}, or \texttt{dcbz} is in storage that is Write Through Required or Caching Inhibited.

If the XER specifies a length of zero for an indexed Move Assist instruction, an Alignment interrupt does not occur.

Setting the DSISR and DAR as described below is optional for implementations on which Alignment interrupts occur rarely, if ever, for cases that the Alignment interrupt handler emulates. For such implementations, if the DSISR and DAR are not set as described below they are set to undefined values.

The following registers are set:

\textbf{SRR0} Set to the effective address of the instruction that caused the interrupt.

\textbf{SRR1}
- \texttt{33:36} Set to 0.
- \texttt{42:47} Set to 0.
- Others Loaded from the MSR.

\textbf{MSR} See Figure 51.

\textbf{DSISR}
- \texttt{32:43} Set to 0.
- \texttt{44:45} Set to bits 30:31 of the instruction if DS-form. Set to 0b00 if D-, or X-form.
- \texttt{46} Set to 0.
- \texttt{47:48} Set to bits 29:30 of the instruction if X-form. Set to 0b00 if D- or DS-form.
- \texttt{49} Set to bit 25 of the instruction if D-form. Set to 0b00 if D- or DS-form.
- \texttt{50:53} Set to bits 21:24 of the instruction if X-form. Set to bits 1:4 of the instruction if D- or DS-form.
- \texttt{54:58} Set to bits 6:10 of the instruction (RT/RS/FRT/FRS), except undefined for \texttt{dcbz}.
- \texttt{59:63} Set to bits 11:15 of the instruction (RA) for update form instructions; set to either bits 11:15 of the instruction or to any register number not in the range of registers to be loaded for a valid form \texttt{lmw}, a valid form \texttt{lswi}, or a valid form \texttt{lswx} for which neither RA nor RB is in the range of registers to be loaded; otherwise undefined.

\textbf{DAR} Set to the effective address computed by the instruction, except that if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

For an X-form \texttt{Load} or \texttt{Store}, it is acceptable for the thread to set the DSISR to the same value that would have resulted if the corresponding D- or DS-form instruction had caused the interrupt. Similarly, for a D- or DS-form \texttt{Load} or \texttt{Store}, it is acceptable for the thread to set the DSISR to the value that would have resulted for the corresponding X-form instruction. For example, an unaligned \texttt{l wax} (that crosses a protection boundary) would normally, following the description above, cause the DSISR to be set to binary:

\begin{verbatim}
0000000000 00 0 01 0 0101 ttttt ??????
\end{verbatim}

where “tttt” denotes the RT field, and “????????” denotes an undefined 5-bit value. However, it is acceptable if it causes the DSISR to be set as for \texttt{lwa}, which is

\begin{verbatim}
0000000000 10 0 00 0 1101 ttttt ??????
\end{verbatim}

If there is no corresponding alternative form instruction (e.g., for \texttt{l wax}), the value described above is set in the DSISR.

The instruction pairs that may use the same DSISR value are:

- \texttt{lhz/lhzx}, \texttt{lhzu/lhzux}, \texttt{lha/lhax}, \texttt{lhma/lhaux}
- \texttt{lwz/lwzx}, \texttt{lwzu/lwzux}, \texttt{lwa/lwax}
- \texttt{id/idx}, \texttt{ldud/ldux}
- \texttt{lsth/sthx}, \texttt{sthu/sthux}
- \texttt{std/stdx}, \texttt{stdu/stdux}
- \texttt{lfs/lfx}, \texttt{lfsu/lfsux}
- \texttt{lfd/lfx}, \texttt{lfdu/lfxu}
- \texttt{sts/strs}, \texttt{stfs/strfs}, \texttt{std/strd}, \texttt{stdf/strfd}

Execution resumes at effective address \texttt{0x0000_0000_0000_0600}, possibly offset as specified in Figure 52.

\textbf{Programming Note}

The architecture does not support the use of an unaligned effective address by \texttt{l harx}, \texttt{l warx}, \texttt{ldarx}, \texttt{sthcx}, \texttt{stwcx}, \texttt{stdcx}, \texttt{eciwx}, and \texttt{ecowx}.

If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to simulate the instruction, but instead should treat the instruction as a programming error.

\subsection{6.5.9 Program Interrupt}

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

\textbf{Floating-Point Enabled Exception}

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

\begin{verbatim}
(MSRFEO | MSRFE1) & FSPCRFEX
\end{verbatim}

is 1. FSPCRFEX is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a \texttt{Move To FSPCR}
instruction that causes an exception bit and the corresponding enable bit both to be 1.

**TM Bad Thing Exception [Category: Transactional Memory]**

A TM Bad Thing exception is generated when any of the following occurs.

- An \texttt{rfebb}, \texttt{rifd}, \texttt{hrfid}, or \texttt{mtmsrd} instruction attempts to cause an illegal state transition (see Section 3.2.2).
- An \texttt{rifd}, \texttt{hrfid}, or \texttt{mtmsrd} instruction attempts to cause a transition to Problem state with an active transaction (Transactional or Suspended state) when TM is disabled by the PCR (\texttt{PCR}_{TM}=1 or \texttt{PCR}_{V2,08}=1).
- An \texttt{rfebb} instruction in Problem state attempts to cause a transition to Transactional or Suspended state when \texttt{PCR}_{TM}=1 (i.e. a latent non-zero TS value was in the BESCR).
- An attempt is made to execute \texttt{trechkpt} in Transactional or Suspended state or when \texttt{TEXASR}_{FS}=0.
- An attempt is made to execute \texttt{tend} in Suspended state.
- An attempt is made to execute \texttt{treclaim} in Non-transactional state.
- An attempt is made to execute a \texttt{mtspr} targeting a TM register in other than Non-transactional state.
- An attempt is made to execute a power saving instruction in Suspended state.

**Privileged Instruction**

The following applies if the instruction is executed when MSR\textsubscript{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an \texttt{mtspr} or \texttt{mfspr} instruction with an SPR field that contains a value having spr\textsubscript{0}=1.

The following applies if the instruction is executed when MSR\textsubscript{HV\ PR} = 0b00.

A Privileged Instruction type Program interrupt is generated when execution is attempted of an \texttt{mtspr} or \texttt{mfspr} instruction with an SPR field that designates an SPR that is accessible by the instruction only when the thread is in hypervisor state, or when execution of a hypervisor-privileged instruction is attempted.

**Programming Note**

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR\textsubscript{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining SRR\textsubscript{149} (the bit in which MSR\textsubscript{PR} was saved by the interrupt).

**Trap**

A Trap type Program interrupt is generated when any of the conditions specified in a Trap instruction is met.

The following registers are set:

**SRR0**

For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception Program interrupt, set as described in the following list.

- If MSR\textsubscript{FE0 \ FE1} = 0b00, FPSCR\textsubscript{FE} = 1, and an instruction is executed that changes MSR\textsubscript{FE0 \ FE1} to a nonzero value, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

**Programming Note**

Recall that all instructions that can alter MSR\textsubscript{FE0 \ FE1} are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If MSR\textsubscript{FE0 \ FE} = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If MSR\textsubscript{FE0 \ FE} = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time FPSCR\textsubscript{FE} was changed from 1 to 0 or of some subsequent instruction.

**SRR1**

- Set to 0.
- Set to 1 for a TM Bad Thing Exception type Program interrupt; otherwise set to 0.
- Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.
Section 6.5.10  Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR\textsubscript{FP}=0.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**:
  - 33:36: Set to 0.
  - 42:47: Set to 0.
- **Others**: Loaded from the MSR.
- **MSR**: See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0800, possibly offset as specified in Figure 52.

Section 6.5.11  Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSR\textsubscript{EE}=1.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **SRR1**:
  - 33:36: Set to 0.
  - 42:47: Set to 0.
- **Others**: Loaded from the MSR.
- **MSR**: See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0900, possibly offset as specified in Figure 52.

Section 6.5.12  Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

\[
\text{MSR}_{\text{EE}} \mid \neg (\text{MSR}_{\text{HV}}) \mid \text{MSR}_{\text{PR}} \mid \text{HDICE}
\]

The following registers are set:

- **HSRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
6.5.13 Directed Privileged Doorbell Interrupt [Category: Server. Processor Control]

A Directed Privileged Doorbell interrupt occurs when no higher priority exception exists, a Directed Privileged Doorbell exception is present, and MSR_EE=1. Directed Privileged Doorbell exceptions are generated when Directed Privileged Doorbell messages (see Chapter 11) are received and accepted by the thread.

Save/Restore Register 0, Save/Restore Register 1 and Machine State Register are updated as follows:

- **Save/Restore Register 0**: Set to the effective address of the next instruction to be executed.
- **Save/Restore Register 1**: Set to the contents of the Machine State Register at the time of the interrupt.
- **Machine State Register**: See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0980, possibly offset as specified in Figure 52.

---

**Programming Note**

Because the value of MSR_EE is always 1 when the thread is in problem state, the simpler expression

\[(MSR_{EE} \lor \neg(MSR_{HV})) \& HDICE\]

is equivalent to the expression given above.

6.5.14 System Call Interrupt

A System Call interrupt occurs when a System Call instruction is executed.

The following registers are set:

- **SRR0**: Set to the effective address of the instruction following the System Call instruction.
- **SRR1**: Set to 0.
- **Others**: Loaded from the MSR.
- **MSR**: See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0C00, possibly offset as specified in Figure 52.

---

**Programming Note**

An attempt to execute an `sc` instruction with LEV=1 in problem state should be treated as a programming error.

6.5.15 Trace Interrupt [Category: Trace]

A Trace interrupt occurs when no higher priority exception exists and any instruction except `rfid`, `hrfid`, or a Power-Saving Mode instruction is successfully completed, provided any of the following is true:

- the instruction is `mtmsr[d]` and MSR_S=1 when the instruction was initiated,
- the instruction is not `mtmsr[d]` and MSR_S=1,
- the instruction is a `Branch` instruction and MSR_BE=1, or
- a CIABR match occurs.

Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state <TM>, did not cause the transaction to fail in such a way that the instruction did not complete. (See Section 5.3.1 of Book II). Thus a Trace interrupt never occurs for a System Call instruction, or for a Trap instruction that traps, or for a `dcbf` that is executed in Transactional state. The instruction that causes a Trace interrupt is called the "traced instruction".

When a Trace interrupt occurs, the following registers are set:

- **SRR0**: Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **SRR1**: Set to 1.
- **Others**: Loaded from the MSR.
6.5.16 Hypervisor Data Storage Interrupt

A Hypervisor Data Storage interrupt occurs when no higher priority exception exists, the thread is not in hypervisor state, and either (a) VPM1=0, LPCRKBV=1, and a Virtual Storage Page Class Key Protection exception exists or (b) the value of the expression

\[(VPM_0 \& \neg MSR_{DR}) \mid (VPM_1 \& MSR_{DR})\]

is 1, and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of the storage location specified by a Load, Store, icbi, dcbz, dcbst, dcbf[l], eciwx, or ecowx instruction cannot be translated to a real address.
- Data address translation is disabled (MSR_{DR}=0), and the virtual address of any byte of the storage location specified by a Load, Store, icbi, dcbz, dcbst, dcbf[l], eciwx, or ecowx instruction cannot be translated to a real address by means of the virtual real addressing mechanism.
- The effective address specified by a lq, stq, lbarx, lharx, lwarx, ldarx, lqarx, stbcx, sthcx, stwcx, stdcx, or stqcx. instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates storage protection.
- A Data Address Watchpoint match occurs.
- Execution of an eciwx or ecowx instruction is disallowed because EARE=0.

If a stbcx, sthcx, stwcx, stdcx, or stqcx. would not perform its store in the absence of a Hypervisor Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional Store to the specified effective address would cause a Hypervisor Data Storage interrupt, it is implementation-dependent whether a Hypervisor Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed Move Assist instruction, a Hypervisor Data Storage interrupt does not occur.

The following registers are set:

| HSRR0 | Set to the effective address of the instruction that caused the interrupt. |
| HSRR1 | 33:36  Set to 0. |
|       | 42:47  Set to 0. |
| Others | Loaded from the MSR. |
| MSR   | See Figure 51. |
| HDSISR | 32  Set to 0. |
|       | 33  Set to 1 if the value of the expression |
|       | (MSR_{DR}) \mid (\neg MSR_{DR} \& VPM_0) |

In general, interrupt handlers can achieve the effect of tracing these instructions.
is 1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34:35
Set to 0.

36
Set to 1 if the access is not permitted by Figure 32 or 33, as appropriate; otherwise set to 0.

37
Set to 1 if the access is due to a \texttt{lsq}, \texttt{stq}, \texttt{ldarx}, \texttt{lwax}, \texttt{ldaxr}, \texttt{lqax}, \texttt{stbcx}, \texttt{sthcx}, \texttt{stwcx}, \texttt{stdcx}, or \texttt{stqcx} instruction that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.

38
Set to 1 if a \texttt{Store}, \texttt{dcbz}, or \texttt{ecowx} instruction; otherwise set to 0.

39:40
Set to 0.

41
Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

42
Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43
Set to 1 if execution of an \texttt{eciwx} or \texttt{ecowx} instruction is attempted when \texttt{EARE}=0; otherwise set to 0.

44:63
Set to 0.

**HDAR**

Set to the effective address of a storage element, as described in the following list. The list should be read from the top down; the HDAR is set as described by the first item that corresponds to an exception that is reported in the HDSISR. For example, if a \texttt{Load Word} instruction causes a storage protection violation and a Data Address Watchpoint match, the HDAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match or, for \texttt{eciwx} and \texttt{ecowx}, \texttt{EARE}=0
  - a byte in the block that caused the exception, for a \texttt{Cache Management} instruction
  - a byte in the first aligned quadword for which access was attempted in the page that caused the exception, for a quadword \texttt{Load} or \texttt{Store} instruction (i.e., a \texttt{Load} or \texttt{Store} instruction for which the storage operand is a quadword; “first” refers to address order; see Section 6.7)
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a non-quadword \texttt{Load} or \texttt{Store} instruction or an \texttt{eciwx} or \texttt{ecowx} instruction
  - undefined, for a Data Address Watchpoint match, or if \texttt{eciwx} or \texttt{ecowx} is executed when \texttt{EARE}=0

For the cases in which the HDAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the HDAR are set to 0.

If multiple Hypervisor Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the HDSISR. If the HDSISR reports other exceptions together with a Virtualized Page Class Key Storage Protection exception that occurs when LPCRKBV=1 and Virtualized Partition Memory is disabled by VPM1=0, the other exceptions are actually DSIs.

---

**Programming Note**

A Virtual Page Class Key Storage Protection exception that occurs with LPCRKBV=1 and Virtualized Partition Memory disabled by VPM1=0 identifies an access that must be emulated by the hypervisor. When it is reported together with other exceptions in the HDSISR, the hypervisor should service the Virtual Page Class Key Storage Protection exception first. This is in part because the operating system may be using some PTE fields for non-architected purposes, which could in turn cause spurious exceptions to be reported.

Execution resumes at effective address 0x0000_0000_0000_0E00, possibly offset as specified in Figure 52.

---

**6.5.17 Hypervisor Instruction Storage Interrupt**

A Hypervisor Instruction Storage interrupt occurs when the thread is not in hypervisor state, no higher priority exception exists, the value of the expression $(VPM_0 \& \neg MSR_0) \lor (VPM_1 \& MSR_1)$ is 1, and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled (MSR$_i$=1) and the virtual address cannot be translated to a real address.
- Instruction address translation is disabled (MSR$_i$=0), and the virtual address cannot be translated to a real address by means of the virtual real addressing mechanism.
- The fetch access violates storage protection.

The following registers are set:
**HSR0** Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, HSR0 is set to the branch target address).

**HSRR1**

33 Set to 1 if the value of the expression \((\text{MSR}_{\text{IP}}) \mid (\neg \text{MSR}_{\text{IP}} \& \text{VP}_{\text{PM}})\) is 1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access is to No-execute or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 32 or 33, as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43:47 Set to 0.

Others Loaded from the MSR.

**MSR** See Figure 51.

If multiple Hypervisor Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in HSRR1.

Execution resumes at effective address 0x0000_0000_0000_0E10, possibly offset as specified in Figure 52.

### 6.5.18 Hypervisor Emulation Assistance Interrupt

A Hypervisor Emulation Assistance interrupt is generated when execution is attempted of an illegal instruction, or of a reserved instruction or an instruction that is not provided by the implementation. It is also generated under the following conditions.

- an **mtepr** or **mtepr** instruction is executed when MSR_{PR}=1 if the instruction specifies an SPR with spr_{0}=0 that is not provided by the implementation
- an **mtepr** or **mtepr** instruction is executed when MSR_{PR}=0 if the instruction specifies SPR 0
- an **mtepr** instruction is executed when MSR_{PR}=0 if the instruction specifies SPR 4, 5, or 6

A Hypervisor Emulation Assistance interrupt may be generated when execution is attempted of any of the following kinds of instruction.

- an instruction that is in invalid form
- an **lsdx** instruction for which RA or RB is in the range of registers to be loaded

The following registers are set:

- **HSR0** Set to the effective address of the instruction that caused the interrupt.
- **HSR1**
  - 33:36 Set to 0.
  - 42:47 Set to 0.
  - Others Loaded from the MSR.
- **MSR** See Figure 51 on page 951.
- **HEIR** Set to a copy of the instruction that caused the interrupt

Execution resumes at effective address 0x0000_0000_0000_0E40, possibly offset as specified in Figure 52.

#### Programming Note

If a Hypervisor Emulation Assistance interrupt occurs, when the thread is not in hypervisor state, for an instruction that the hypervisor does not emulate, the hypervisor should pass control to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt", as described in a Programming Note near the end of Section 6.5.9, "Program Interrupt" on page 959.

### 6.5.19 Hypervisor Maintenance Interrupt

A Hypervisor Maintenance interrupt occurs when no higher priority exception exists, a Hypervisor Maintenance exception exists (a bit in the HMER is set to one), the exception is enabled in the HMEER, and the value of the following expression is 1.

\((\text{MSR}_{\text{EE}} \mid \neg (\text{MSR}_{\text{HV}}) \mid \text{MSR}_{\text{PR}})\)

The following registers are set:

- **HSR0** Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.
- **HSRR1**
  - 33:36 Set to 0.
  - 42:47 Set to 0.
  - Others Loaded from the MSR.
- **MSR** See Figure 51 on page 951.
- **HMER** See Section 6.2.8 on page 940.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an **mthmtr** instruction.

Execution resumes at effective address 0x0000_0000_0000_0E60.
6.5.20 Directed Hypervisor Doorbell Interrupt [Category: Server.Processor Control]

A Directed Hypervisor Doorbell interrupt occurs when no higher priority exception exists, a Directed Hypervisor Doorbell exception is present, and MSREE=0 | MSRHV=1. Directed Hypervisor Doorbell exceptions are generated when Directed Hypervisor Doorbell messages (see Chapter 11) are received and accepted by the thread.

The following registers are set:
- **SRR0**: Set to the effective address of the instruction that would have been attempted to be executed next if no interrupt conditions were present.
- **SRR1**: 33:36 and 42:47 set to 0. 43:47 set to 0. Others loaded from the MSR.
- **MSR**: See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0E80, possibly offset as specified in Figure 52.

6.5.22 Vector Unavailable Interrupt [Category: Vector]

A Vector Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a Vector instruction (including Vector loads, stores, and moves), and MSRVEC=0.

The following registers are set:
- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: 33:36 set to 0. 42:47 set to 0. Others loaded from the MSR.
- **MSR**: See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0F20, possibly offset as specified in Figure 52.

6.5.23 VSX Unavailable Interrupt [Category: VSX]

A VSX Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a VSX instruction (including VSX loads, stores, and moves), and MSRVSX=0.

The following registers are set:
- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: 33:36 set to 0. 42:47 set to 0. Others loaded from the MSR.
- **MSR**: See Figure 51 on page 951.
Execution resumes at effective address 0x0000_0000_0000_0F40, possibly offset as specified in Figure 52.

6.5.24 Facility Unavailable Interrupt

A Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem state when it has been made unavailable by the FSCR.
- a Performance Monitor register or instruction (e.g. mtbhrbe, clrbhrb) is accessed in problem state when it has been made unavailable by MMCR0.
- the Transactional Memory Facility is accessed in any privilege state when it has been made unavailable by MSR_TM.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRR0</td>
<td>Set to the effective address of the instruction that caused the interrupt.</td>
</tr>
<tr>
<td>SRR1</td>
<td>Set to 0.</td>
</tr>
<tr>
<td></td>
<td>33:36 Set to 0.</td>
</tr>
<tr>
<td></td>
<td>42:47 Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
<tr>
<td>MSR</td>
<td>See Figure 51 on page 951.</td>
</tr>
<tr>
<td>FSCR</td>
<td>Set to the effective address 0x0000_0000_0000_0F60, possibly offset as specified in Figure 52.</td>
</tr>
<tr>
<td></td>
<td>0:7 See Section 6.2.10 on page 941.</td>
</tr>
<tr>
<td></td>
<td>Others Not changed.</td>
</tr>
</tbody>
</table>

Execution resumes at effective address 0x0000_0000_0000_0F80, possibly offset as specified in Figure 52.

Programming Note

For the case of an outer tbegin, the interrupt handler should either return to the tbegin, with MSR_TM = 1 (allowing the program to use transactions), or treat the attempt to initiate an outer transaction as a program error.

6.5.25 Hypervisor Facility Unavailable Interrupt

A Hypervisor Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem or privileged states when it has been made unavailable by the HFSCR.

The following registers are set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSRR0</td>
<td>Set to the effective address of the instruction that caused the interrupt.</td>
</tr>
<tr>
<td>HSRR1</td>
<td>Set to 0.</td>
</tr>
<tr>
<td>Others</td>
<td>Loaded from the MSR.</td>
</tr>
<tr>
<td>MSR</td>
<td>See Figure 51 on page 951.</td>
</tr>
<tr>
<td>HFSCR</td>
<td>0:7 See Section 6.2.11 on page 942.</td>
</tr>
<tr>
<td>Others</td>
<td>Not changed.</td>
</tr>
</tbody>
</table>

Execution resumes at effective address 0x0000_0000_0000_0F80, possibly offset as specified in Figure 52.
6.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a Load or Store instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects:

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a Load instruction nor modified by a Store instruction. Also, the rules for storage accesses given in Section 5.8.1, "Guarded Storage" and in Section 2.1 of Book II are obeyed.
- Some registers may have been altered as described in the Book II section cited above.
- Reference and Change bits may have been updated as described in Section 5.7.8.
- For a stbcx, sthcx, stwcx, stdcx, or stdqcx instruction that is executed in-order, CR0 may have been set to an undefined value and the reservation may have been cleared.

The architecture does not support continuation of an aborted instruction but intends that the aborted instruction be re-executed if appropriate.

---

**Programming Note**

An exception may result in the partial execution of a Load or Store instruction. For example, if the Page Table Entry that translates the address of the storage operand is altered, by a program running on another thread, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the Load or Store instruction to be aborted after having been partially executed.

As stated in the Book II section cited above, if an instruction is partially executed the contents of registers are preserved to the extent that the instruction can be re-executed correctly. The consequent preservation is described in the following list. For any given instruction, zero, one, or two items in the list apply.

- For a fixed-point Load instruction that is not a multiple or string form, or for an eciwx instruction, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an lq instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form Load or Store instruction, the contents of register RA are not altered.
6.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the Mediated External exception, persist and can be deferred. However, other exceptions would be lost if they were not recognized and handled when they occur. For example, if an External interrupt was generated when a Data Storage exception existed, the Data Storage exception would be lost. If the Data Storage exception was caused by a Store Multiple instruction for which the storage operand crosses a virtual page boundary and the exception was a result of attempting to access the second virtual page, the store could have modified locations in the first virtual page even though it appeared that the Store Multiple instruction was never executed.

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

Data Storage, Hypervisor Data Storage, Data Segment, and Alignment exceptions and transaction failure due to attempted access of a disallowed type while in Transactional state occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0). (The required ordering of exceptions on components of non-atomic accesses does not extend to the performing of the component accesses in the event of an exception. For example, if byte n causes a data storage exception, it is not necessarily true that the access to byte n-1 has been performed.)

6.7.1 Unordered Exceptions

The exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented.

1. System Reset
2. Machine Check

6.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism. With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same ordering. The exception is that Virtual Page Class Key Storage Protection exceptions that occur when LPCR KEY=1 and Virtualized Partition Memory is disabled by VPM1=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

System-Caused or Imprecise

1. Program
    1. Imprecise Mode Floating-Point Enabled Exception
2. Hypervisor Maintenance
Instruction-Caused and Precise

1. Instruction Segment
2. [Hypervisor] Instruction Storage
3.a Hypervisor Emulation Assistance
3.b Program
   - Privileged Instruction
4. Function-Dependent
   4.a Fixed-Point and Branch
      1 Hypervisor Facility Unavailable
      2 Facility Unavailable
      3a Program
         - Trap
      3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      3c.1 Data Storage for the case of Fixed-Point Load or Store Caching Inhibited instructions with MSR DR=1
      3c.2 all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, or Alignment
4 Trace
4.b Floating-Point
   1 Hypervisor Facility Unavailable
   2 FP Unavailable
   3a Program
      - Precise Mode Floating-Point Enabled Exception
   3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
4 Trace
4.c Vector
   1 Hypervisor Facility Unavailable
   2 Vector Unavailable
   3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
4 Trace
4.d VSX
   1 Hypervisor Facility Unavailable
   2 VSX Unavailable
   3a Program
      - Precise Mode Floating-Point Enabled Exception
   3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
4 Trace
4.e Other Instructions
   1 Hypervisor Facility Unavailable
   2 Facility Unavailable
   3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
4 Trace

Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction. The External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell <S.PC>, and Directed Hypervisor Doorbell <S.PC> interrupts have equal ordering. Similarly, where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal ordering.

Even on threads that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

6.8 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority. The phrase "corresponding interrupt" means the interrupt having the same name as the exception unless the thread is in power-saving mode, in which case the phrase means the System Reset interrupt.

Unless otherwise stated or obvious from context, it is assumed below that one of the following conditions is satisfied.

- The thread is not in power-saving mode and the interrupt, unless it is the Machine Check interrupt, is not disabled. (For the Machine Check interrupt no assumption is made regarding enablement.)
- The thread is in power-saving mode and the exception is enabled to cause exit from the mode.

With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same priority. The exception is that exceptions caused by Virtual Page Class Key Storage Protection exceptions that occur when LPCR KBV=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

1. System Reset

System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.
Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check

Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt.

Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Caused and Precise

This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where [Hypervisor] Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the hardware may generate any one of the three interrupts for which an exception exists). For instructions that are forbidden in Transactional state, transaction failure takes priority over all interrupts except Privileged Instruction type of Program Interrupts. For data accesses that are forbidden in Transactional state, transaction failure has the same priority as the group of “other” [Hypervisor] Data Storage, Data Segment, and Alignment exceptions. (See Section 5.3.1 of Book II).

A. Fixed-Point Loads and Stores
   a. These exceptions are mutually exclusive and have the same priority:
      ■ Hypervisor Emulation Assistance
      ■ Program - Privileged Instruction
   b. Hypervisor Facility Unavailable
   c. Facility Unavailable
   d. Data Storage for the case of Fixed-Point Load or Store Caching Inhibited instructions with MSRDR=1
   e. All other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, or Alignment
   f. Trace

B. Floating-Point Loads and Stores
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Floating-Point Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
   e. Trace

C. Vector Loads and Stores
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Vector Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
   e. Trace

D. VSX Loads and Stores
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. VSX Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
   e. Trace

E. Other Floating-Point Instructions
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Floating-Point Unavailable
   d. Program - Precise Mode Floating-Point Enabled Exception
   e. Trace

F. Other Vector Instructions
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. Vector Unavailable
   d. Trace

G. Other VSX Instructions
   a. Hypervisor Emulation Assistance
   b. Hypervisor Facility Unavailable
   c. VSX Unavailable
   d. Program - Precise Mode Floating-Point Enabled Exception
   e. Trace

H. TM instruction, mtfspr specifying TM SPR
   a. Program - Privileged Instruction (only for treclaim, trechkpt, and mtsp)
   b. Hypervisor Facility Unavailable
   c. Facility Unavailable
   d. Program - TM Bad Thing (only for treclaim, trechkpt, and mtsp)
   e. Trace

I. rfd, hrfid, rfmb and mtmsr[d]
   a. Program - Privileged Instruction for all except rfebb
   b. Program - TM Bad Thing exception for all except mtsp
   c. Program - Floating-Point Enabled Exception for all except rfebb
   d. Trace, for mtmsr[d] and rfebb only

J. Other Instructions
   a. These exceptions are mutually exclusive and have the same priority:
      ■ Program - Trap
System Call
Program - Privileged Instruction
Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. Facility Unavailable
d. Trace

K. [Hypervisor] Instruction Storage and Instruction Segment

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. Hypervisor Maintenance

This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Maintenance exception exists and each attempt to execute an instruction when the Hypervisor Maintenance interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Maintenance interrupt is not delayed indefinitely.

If LPES=1 and a Direct External exception exists and each attempt to execute an instruction when this interrupt is enabled causes an exception (see the Programming Note below), the Direct External interrupt is not delayed indefinitely.

6.9 Relationship of Event-Based Branches to Interrupts

Event-based exceptions have a priority lower than all exceptions that cause interrupts. When an event-based exception is created, the Event-Based Branch facility waits for all other possible exceptions that would cause interrupts to be reported. It then generates the event-based branch if no exception that would cause an interrupt exists when the event-based branch is to be generated.

---

Programming Note

An incorrect or malicious operating system could corrupt the first instruction in the interrupt vector location for an instruction-caused interrupt such that the attempt to execute the instruction causes the same exception that caused the interrupt (a looping interrupt; e.g., Trap instruction and Program interrupt). Similarly, the first instruction of the interrupt vector for one instruction-caused interrupt could cause a different instruction-caused interrupt, and the first instruction of the interrupt vector for the second instruction-caused interrupt could cause the first instruction-caused interrupt (e.g., Program interrupt and Floating-Point Unavailable interrupt). Similarly, if the Real Mode Area is virtualized and there is no PTE for the page containing the interrupt vectors, every attempt to execute the first instruction of the OS's Instruction Storage interrupt handler would cause a Hypervisor Instruction Storage interrupt; if the Hypervisor Instruction Storage interrupt handler returns to the OS’s Instruction Storage interrupt handler without the relevant PTE having been created, another Hypervisor Instruction Storage interrupt would occur immediately. The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.
Chapter 7. Timer Facilities

7.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, Processor Utilization of Resources, and Scaled Processor Utilization of Resources registers provide timing functions for the system. The remainder of this section describes these registers and related facilities.

7.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 53) containing a 64-bit unsigned integer that is incremented periodically.

![Figure 53. Time Base](image)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBU40</td>
<td>Upper 40 bits of Time Base</td>
</tr>
<tr>
<td>TBU</td>
<td>Upper 32 bits of Time Base</td>
</tr>
<tr>
<td>TBL</td>
<td>Lower 32 bits of Time Base</td>
</tr>
</tbody>
</table>

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in problem state (MSRPR=1). If the means is under software control, it must be privileged and, in implementations of the Server environment, must be accessible only in hypervisor state (MSRHVPRI = 0b10). There must be a method for getting all Time Bases in the system to start incrementing with values that are identical or almost identical.

2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Power ISA system. The Time Base update frequency is not required to be constant. What is required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

See Chapter 6 of Book II for information about the update frequency of the Time Base.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.
7.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Chapter 6 of Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The `mttbl` and `mttbu` extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the `mtspr` instruction; see Appendix A, “Assembler Extended Mnemonics” on page 1015.

The Time Base can be written by a sequence such as:

```
lwz     Rx,upper # load 64-bit value for
lwx     Ry,lower # TB into Rx and Ry
li      Rz,0
mttbl   Rx  # set TBL to 0
mttbu   Rx  # set TBU
mttbl   Ry  # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

The preferred method of changing the Time Base utilizes the TBU40 facility. The following code sequence demonstrates the process. Assume the upper 40 bits of

```
lwz Ry,upper # load 64-bit Time Base value
clrldi Ry,Ry,40# lower 24 bits of old TB
mttbu40 Rx # write upper 40 bits of TB
mttb Rz  # read TB value again
clrldi Rz,Rz,40# lower 24 bits of new TB
cmpeq Rz,Ry # compare new and old lwr 24
bge done  # no carry out of low 24 bits
addis Rx,Rx,0x0100#increment upper 40 bits
mttbu40 Rx # update to adjust for carry
```

7.3 Virtual Time Base

The Virtual Time Base (VTB) is a 64-bit incrementing counter.

![Figure 54. Virtual Time Base](image)

The Virtual Time Base increments at the same rate as the Time Base until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^64 - 1); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The operation of the Virtual Time Base has the following additional properties.

1. Loading a GPR from the Virtual Time Base has no effect on the accuracy of the Virtual Time Base.
2. Copying the contents of a GPR to the Virtual Time Base replaces the contents of the Virtual Time Base with the contents of the GPR.

In systems that change the Time Base update frequency for purposes such as power management, the Virtual Time Base input frequency will also change. Software must be aware of this in order to set interval timers.
7.4 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

The Decrementer counts down until its value becomes 0x0000_0000; at the next decrement its value becomes 0xFFFF_FFFF.

The Decrementer is driven at the same frequency as the Time Base.

When the contents of DEC32 change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time.

When the contents of DEC32 change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC32 is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an mtspr instruction.

The operation of the Decrementer has the following additional properties.

1. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

7.5 Hypervisor Decrementer

The Hypervisor Decrementer (HDEC) is a 32-bit decrementing counter that provides a mechanism for causing a Hypervisor Decrementer interrupt after a programmable delay. The contents of the Hypervisor Decrementer are treated as a signed integer.

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

### Programming Note

In configurations in which the hypervisor allows multiple partitions to time-share a processor, the Virtual Time Base can be managed by the hypervisor such that it appears to each partition as if it counts only during the times that the partition is executing.

In order to do this, the hypervisor saves the value of the Virtual Time Base as part of the program context when removing a partition from the processor, and restores it to it's previous value when initiating the partition again on the same or another processor.
menter interrupt occurs, the existing Hypervisor Decrementer exception will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event. Even if multiple HDEC32 transitions from 0 to 1 occur before a Hypervisor Decrementer interrupt occurs, at most one Hypervisor Decrementer exception exists.

The preceding paragraph applies regardless of whether the change in the contents of HDEC32 is the result of decrementation of the Hypervisor Decrementer by the hardware or of modification of the Hypervisor Decrementer caused by execution of an mtspr instruction.

The operation of the Hypervisor Decrementer has the following additional properties.

1. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.

2. Copying the contents of a GPR to the Hypervisor Decrementer replaces the contents of the Hypervisor Decrementer with the contents of the GPR.

---

**Programming Note**

In systems that change the Time Base update frequency for purposes such as power management, the Hypervisor Decrementer update frequency will also change. Software must be aware of this in order to set interval timers.

If Hypervisor Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

---

**Programming Note**

A Hypervisor Decrementer exception is not created if the thread is in a power-saving mode when HDEC32 changes from 0 to 1 because having a Hypervisor Decrementer interrupt occur almost immediately after exiting the power-saving mode in this case is deemed unnecessary. The hypervisor already has control, and if a timed exit from the power-saving mode is necessary and possible, the hypervisor can use the Decrementer to exit the power-saving mode at the appropriate time. For sleep and rwalk power-saving levels, the state of the Hypervisor Decrementer and Decrementer is not necessarily maintained and updated.

---

### 7.6 Processor Utilization of Resources Register (PURR)

The Processor Utilization of Resources Register (PURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the PURR are treated as a 64-bit unsigned integer.

<table>
<thead>
<tr>
<th>PURR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

**Figure 57. Processor Utilization of Resources Register**

The PURR is a hypervisor resource; see Chapter 2.

The contents of the PURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^64 - 1) at which point the contents are replaced by that sum modulo 2^64. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the PURR increases is an estimate of the portion of resources used by the thread per unit time with respect to other threads that share those resources monitored by the PURR. When the thread is idle, the rate at which the PURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the difference between the value represented by the contents of the PURR at time T_a and T_b be the value P_{ab}. The ratio of P_{ab}/T_{ab} is an estimate of the percentage of shared resources used by the thread during the interval T_{ab}. For the set (S) of threads that share the resources monitored by the PURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the PURR are implementation-specific.

The PURR is implemented such that:

1. Loading a GPR from the PURR has no effect on the accuracy of the PURR.

2. Copying the contents of a GPR to the PURR replaces the contents of the PURR with the contents of the GPR.
7.7 Scaled Processor Utilization of Resources Register (SPURR)

The Scaled Processor Utilization of Resources Register (SPURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the SPURR are treated as a 64-bit unsigned integer.

![Figure 58. Scaled Processor Utilization of Resources Register](image)

The SPURR is a hypervisor resource; see Section 2.7.

The contents of the SPURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^64 - 1) at which point the contents are replaced by that sum modulo 2^64. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the SPURR increases is an estimate of the portion of resources used by the thread with respect to other threads that share those resources monitored by the SPURR, and relative to the computational capacity provided by those resources. The computational capacity provided by the shared resources may vary as a function of the frequency of the oscillator which drives the resources or as a result of deliberate delays in processing that are created to reduce power consumption. When the thread is idle, the rate at which the SPURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be the value S_{ab}. The ratio of S_{ab}/(T_{ab} \times f_e \times (1 - c_r)) is an estimate of the percentage of shared resource capacity used by the thread during the interval T_{ab}. For the set \{S\} of threads that share the resources monitored by the SPURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the SPURR are implementation-specific.

The SPURR is implemented such that:

1. Loading a GPR from the SPURR has no effect on the accuracy of the SPURR.
2. Copying the contents of a GPR to the SPURR replaces the contents of the SPURR with the contents of the GPR.

![Programming Note](image)

Estimates computed as described above may be useful for purposes of resource use accounting, program dispatching, etc.

7.8 Instruction Counter

The Instruction Counter (IC) is a 64-bit incrementing counter that counts the number of instructions that the thread has completed (according to the sequential execution model; see Chapter 2.2 of Book I).

![Figure 59. Instruction Counter](image)
Chapter 8. Debug Facilities

8.1 Overview

Implementations provide debug facilities to enable hardware and software debug functions, such as control flow tracing, data address watchpoints, and program single-stepping. The debug facilities described in this section consist of the Come-From Address Register (see Section 8.1.1), Completed Instruction Address Breakpoint Register (see Section 8.1.2), and the Data Address Watchpoint Register (DAWRn) and Data Address Watchpoint Register Extension (DAWRXn) (see Section 8.1.3). The interrupt associated with the Data Address Breakpoint registers is described in Section 6.5.3. The interrupt associated with the Completed Instruction Address Breakpoint Register is described in Section 6.5.15. The Trace facility, which can be used for single-stepping as well as for control flow tracing, is described in Section 6.5.15.

The mfspr and mtspw instructions (see Section 4.4.4) provide access to the registers of the debug facilities. In addition to the facilities mentioned above, implementations typically provide debug facilities, modes, and access mechanisms that are implementation-specific. For example, implementations typically provide facilities for instruction address tracing, and also access to certain debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

8.1.1 Come-From Address Register

The Come-From Address Register (CFAR) is a 64-bit register. When an rfebb, rfid, instruction is executed, the register is set to the effective address of the instruction. When a Branch instruction is executed and the branch is taken, the register is set to the effective address of an instruction in the instruction cache block containing the Branch instruction, except that if the Branch instruction is a B-form Branch (i.e. bc, bca, bcl, or bcla) for which the target address is in the instruction cache block containing the Branch instruction or is in the previous or next cache block, the register is not necessarily set. For Branch instructions, the setting need not occur until a subsequent context synchronizing operation has occurred.

Figure 60. Come-From Address Register

The contents of the CFAR can be read and written using the mfspr and mtspw instructions. Access to the CFAR is privileged.

Programming Note

This register can be used for purposes of debugging software. For example, often a software bug results in the program executing a portion of the code that it should not have reached or causing an unexpected interrupt. In the former case, a breakpoint can be placed in the portion of the code that was erroneously reached and the program reexecuted. In either case, the interrupt handler can save the contents of the CFAR (before executing the first instruction that would modify the register), and then make the saved contents available for a debugger to use in determining the control flow path by which the exception was reached.

In order to preserve the CFAR's contents for each partition and to prevent it from being used to implement a "covert channel" between partitions, the hypervisor should initialize/save/restore the CFAR when switching partitions on a given thread.

8.1.2 Completed Instruction Address Breakpoint [Category: Trace]

The Completed Instruction Address Breakpoint mechanism provides a means of detecting an instruction completion at a specific instruction address. The address comparison is done on an effective address (EA).

The Completed Instruction Address Breakpoint mechanism is controlled by the Completed Instruction
Address Breakpoint Register (CIABR), shown in Figure 62.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:61</td>
<td>CIEA</td>
<td>Completed Instruction Effective Address</td>
</tr>
<tr>
<td>62:63</td>
<td>PRIV</td>
<td>Privilege</td>
</tr>
</tbody>
</table>

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A Completed Instruction Address Breakpoint match causes a Trace exception provided that no higher priority interrupt occurs from the completion of the instruction (see Section 6.5.15).

8.1.3 Data Address Watchpoint

The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to a range of addresses starting at a designated doubleword. The address comparison is done on an effective address (EA).

The Data Address Watchpoint mechanism is controlled by a single set of SPRs, numbered with n=0: the Data Address Watchpoint Register (DAWRn), shown in Figure 62, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 63.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:60</td>
<td>DEAW</td>
<td>Data Effective Address Watchpoint</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48:53</td>
<td>MRD</td>
<td>Match Range in Doublewords biased by -1. (0b0000000 = 1 DW, 0b111111 = 64 DW)</td>
</tr>
<tr>
<td>56</td>
<td>HRAMMC</td>
<td>Hypervisor Real Addressing Mode Match Control</td>
</tr>
<tr>
<td>57</td>
<td>DW</td>
<td>Data Write</td>
</tr>
<tr>
<td>58</td>
<td>DR</td>
<td>Data Read</td>
</tr>
<tr>
<td>59</td>
<td>WT</td>
<td>Watchpoint Translation</td>
</tr>
<tr>
<td>60</td>
<td>WTI</td>
<td>Watchpoint Translation Ignore</td>
</tr>
<tr>
<td>61:63</td>
<td>PRIVM</td>
<td>Privilege Mask</td>
</tr>
<tr>
<td>62</td>
<td>PNH</td>
<td>Privileged but Non-Hypervisor state</td>
</tr>
<tr>
<td>63</td>
<td>PRO</td>
<td>Problem state</td>
</tr>
</tbody>
</table>

All other fields are reserved.

The supported PRIVM values are 0b000, 0b001, 0b101, and 0b110. If the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.

--- Architecture Note ---

Future versions of the architecture may allow n to be nonzero, thus providing more Data Address Watchpoints. The SPR numbers for DAWRn and DAWRXn have been defined to support this possible future expansion.

--- Programming Note ---

PRIVM value 0b000 causes matches not to occur regardless of the contents of other DAWRn and DAWRXn fields. PRIVM values 0b101 and 0b110 are not supported because a storage location that is shared between the hypervisor and non-hypervisor software is unlikely to be accessed using the same EA by both the hypervisor and the non-hypervisor software. (PRIVM value 0b111 is supported primarily for reasons of software compatibility with respect to emulation of the DABR facility as described in a subsequent Programming Note.)

A Data Address Watchpoint match occurs for a Load or Store instruction if, for any byte accessed, all of the following conditions are satisfied.

- the access is
a quadword access and located in the range
\((DEAW_0:59 \ || \ 0) \leq (EA_0:59 \ || 0) \leq ((DEAW_0:59 \ || 0) + (550 \ || MRD_0:5)))\) such that
\((EA_0:60 \ AND (551 \ || 60)) = (DEAW_0:60 \ AND (551 \ || 60)).\)
- not a quadword access and located in the range \(DEAW_0:60 \leq EA_0:60 \leq (DEAW_0:60 + (550 \ || MRD_0:5)))\) such that
\((EA_0:60 \ AND (551 \ || 60)) = (DEAW_0:60 \ AND (551 \ || 60)).\)
- \((MSR_{DR} = DAWRX_nVT) \ | \ DAWRXnWTI\)
- the thread is in
  - hypervisor state and DAWRXnHYP = 1, or
  - privileged but non-hypervisor state and DAWRXnPHN = 1, or
  - problem state and DAWRXnPR = 1
- the instruction is a \textit{Store} and DAWRXnDW = 1, or
  the instruction is a \textit{Load} and DAWRXnDR = 1.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

If the above conditions are satisfied, a match also occurs for \textit{eciw} and \textit{ecow}. For the purpose of determining whether a match occurs, \textit{eciw} is treated as a \textit{Load}, and \textit{ecow} is treated as a \textit{Store}.

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is \textit{Store Conditional} but the store is not performed
- The instruction is \textit{dcbz}. (For the purpose of determining whether a match occurs, \textit{dcbz} is treated as a \textit{Store}.)

The \textit{Cache Management} instructions other than \textit{dcbz} never cause a match.

A Data Address Watchpoint match causes a Data Storage exception or a Hypervisor Data Storage exception (see Section 6.5.3, “Data Storage Interrupt” on page 955 and Section 6.5.16, “Hypervisor Data Storage Interrupt” on page 963). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a \textit{Store} or \textit{ecow} instruction causes the match, the storage operand is not modified if the instruction is one of the following:
- any \textit{Store} instruction that causes an atomic access
- \textit{ecow}

**Architecture Note**

For a quadword access, it is possible that the address reported in the DAR/HDAR can point to a byte outside of the range of the watchpoint if the starting address of the watchpoint is not quadword aligned.

**Programming Note**

The Data Address Watchpoint mechanism does not apply to instruction fetches.

**Architecture Note**

For a quadword access, it is possible that the address reported in the DAR/HDAR can point to a byte outside of the range of the watchpoint if the starting address of the watchpoint is not quadword aligned.

**Programming Note**

Implementations that comply with versions of the architecture that precede Version 2.02 do not provide the DABRX (now replaced by DAWRXn). Forward compatibility for software that was written for such implementations (and uses the Data Address Breakpoint facility) can be obtained by setting DAWRXn60:63 to 0b0111.
Chapter 9. Performance Monitor Facility

9.1 Overview

The Performance Monitor facility provides a means of collecting information about program and system performance.

9.2 Performance Monitor Operation

The Performance Monitor facility is controlled by the following features (described in detail in subsequent sections).

- an MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring

- registers
  - PMC1 - PMC6 (Performance Monitor Counters 1 - 6), which count events
  - MMCR0, MMCR1, MMCR2, and MMCRA (Monitor Mode Control Registers 0, 1, 2, and A), which control the Performance Monitor facility
  - SIAR, SDAR, and SIER (Sampled Instruction Address Register, Sampled Data Address Register, and Sampled Instruction Event Register), which contain the address of the “sampled instruction” and of the “sampled data,” and additional information about the “sampled instruction” (see Section 9.4.8 - Section 9.4.10).

- The Branch History Rolling Buffer (BHRB), which is a buffer that contains the target addresses of most recent branch instructions for which the branch was taken.

- the performance monitor interrupt, which can be caused by monitored conditions and events.

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.

- A “counter negative condition” exists when the value in a PMC is negative (i.e., when bit 0 of the PMC is 1). A “Time Base transition event” occurs when a selected bit of the Time Base changes from 0 to 1 (the bit is selected by a field in MMCR0). The term “condition or event” is used as an abbreviation for “counter negative condition or Time Base transition event”. A condition or event can be caused implicitly by the hardware (e.g., incrementing a PMC) or explicitly by software (mtpsr).

- A condition or event is enabled if the corresponding “Enable” bit (i.e. PMC1CE, PMCjCE, or TBEE) in MMCR0 is 1. The occurrence of an enabled condition or event can have side effects within the Performance Monitor, such as causing the PMCs to cease counting.

- An enabled condition or event causes a Performance Monitor alert if Performance Monitor alerts are enabled by the corresponding “Enable” bit in MMCR0. A single Performance Monitor alert may reflect multiple enabled conditions and events.

- When a Performance Monitor alert occurs, MMCR0PMAS is set to 1 and the writing of BHRB entries, if in process, is suspended.

  When the contents of MMCR0PMAS change from 0 to 1, a Performance Monitor exception will come into existence within a reasonable period of time. When the contents of MMCR0PMAS change from 1 to 0, the existing Performance Monitor exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

- A Performance Monitor exception causes one of the following.
  - If MSR_CEE = 1 and MMCR0EBE = 0, an interrupt occurs.
  - If MSR_CEE = 1, MMCR0EBE = 1, a Performance Monitor event-based exception occurs if BESC- CPRME=1, provided FSCREBB and HFSCREBB are enabled. When a Performance Monitor event-based exception occurs,
an event-based branch is generated if BES-CRGE=1.

**Programming Note**
The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor exceptions do not occur) by setting MMCR0 to 0x0000_0000_8000_0000.

### 9.3 Probe No-op Instruction

A probe no-op is an `and 0,0,0` instruction. This form of `and` is reserved for use exclusively by the Performance Monitor.

**Programming Note**
Software can insert probe no-op instructions at various points in a program and configure the performance monitor such that the instruction is eligible for sampling. See Section 9.4.7.

### 9.4 Performance Monitor Facility Registers

The Performance Monitor registers count events, control the operation of the Performance Monitor, and provide associated information.

The elapsed time between the execution of an instruction and the time at which events due to that instruction have been reflected in Performance Monitor registers is not defined. No means are provided by which software can ensure that all events due to preceding instructions have been reflected in Performance Monitor registers. Similarly, if the events being monitored may be caused by operations that are performed out-of-order, no means are provided by which software can prevent such events due to subsequent instructions from being reflected in Performance Monitor registers. Thus the contents obtained by reading a Performance Monitor register may not be precise: it may fail to reflect some events due to instructions that precede the `mfspr` and may reflect some events due to instructions that follow the `mfspr`. This lack of precision applies regardless of whether the state of the thread is such that the register is subject to change by the hardware at the time the `mfspr` is executed. Similarly, if an `mfspr` instruction is executed that changes the contents of the Time Base, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed.

If an `mfspr` instruction is executed that changes the value of a Performance Monitor register other than SIAR, SDAR, and SIER, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009).

**Programming Note**
Depending on the events being monitored, the contents of Performance Monitor registers may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

#### 9.4.1 Performance Monitor SPR Numbers

The Performance Monitor registers have two sets of SPR numbers, one set that is non-privileged and another set that is privileged.

For the purpose of explanation elsewhere in the architecture, the non-privileged registers are divided into two groups as defined below.

- **A**: The non-privileged read/write Performance Monitor registers (i.e. the PMCs, MMCR0, MMCR2, and MMCRA at SPR numbers 771-776, 779, 769, and 770, respectively)
- **B**: The non-privileged read-only Performance Monitor registers (i.e. SIER, SIAR, SDAR, and MMCR1 at SPR numbers 768, 780, 781, and 782, respectively).

The SPRs in group B are treated as not implemented registers for write (`mtspr`) operations. See the `mtspr` instruction description in Section 4.4.4 for additional information.

When the PCR makes a register in either group A or B unavailable in problem state, that SPR is not included in group A or B.

#### 9.4.2 Performance Monitor Counters

The six Performance Monitor Counters, PMC1 through PMC6, are 32-bit registers that count events.

![Performance Monitor Counter registers](image)

**Figure 64. Performance Monitor Counter registers**

PMC1 - PMC4 are referred to as “programmable” counters since the events that can be counted can be...
specified by the program. The codes that identify the events that are counted are selected by specifying the appropriate code in “PMCn Selector” fields in MMCR1. Some events may include operations that are performed out-of-order.

PMC5 and PMC6 are not programmable. PMC5 counts instructions completed and PMC6 counts cycles. The PMCC field in MMCR0 control whether or not PMCs 5-6 are under the control of various bits in MMCR0 and MMCR2. When PMCs 5-6 are not under the control of these bits, they do not cause performance monitor events.

Normally each PMC is incremented each hardware cycle by the number of times the corresponding event occurred in that cycle. Other modes of incrementing may also be provided.

**Programming Note**

PMC5 and PMC6 are defined to facilitate calculating basic performance metrics such as cycles per instruction (CPI).

Software can use a PMC to “pace” the collection of Performance Monitor data. For example, if it is desired to collect event counts every n cycles, software can specify that a particular PMC count cycles and set that PMC to 0x8000_0000 - n. The events of interest would be counted in other PMCs. The counter negative condition that will occur after n cycles can, with the appropriate setting of MMCR bits, cause counter values to become frozen, cause a Performance Monitor exception to occur, etc.

### 9.4.3 Threshold Event Counter

The threshold event counter and associated controls are in MMCRA (see Section 9.4.7). When the Performance Monitor is enabled (MMCR0PMAE=1), this counter begins incrementing from value 0 upon each occurrence of the event specified in the Threshold Event (TEV) field after recognizing the event specified by the Threshold Start (TS) field. The counter stops incrementing when the event specified in the Threshold End (TE) field occurs. The counter subsequently freezes until the event specified in the TS is again recognized, at which point it increments as explained above. Incrementing stops when a Performance Monitor Alert occurs. After the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets MMCR0PMAE to 1.

**Programming Note**

Because hardware can modify the contents of the threshold event counter when random sampling is enabled (MMCRASEM=1) and MMCR0PMAE=1 at any time, any value written to the threshold event counter under this condition may be immediately overwritten by hardware.

The threshold counter value is represented as a base-4 floating-point number. The mantissa of the floating-point number is contained in MMCRA TECM, and the exponent contained in MMCRA TECX. For a given counter mantissa, m, and exponent, e, the number represented is as follows:

\[ N = 4^e \times m \]

This counter format allows the counter to represent a range of 0 through approximately 2M counts with many fewer bits than would be required by a binary counter.

**Programming Note**

Software can obtain the number N from contents of the threshold counter by shifting the mantissa left twice times the value contained in the exponent.

The value in the counter is the exact number of events that occur for values from 0 through the maximum mantissa value (127), within 4 events of the exact value for values from 128 - 508 (or 127^4), within 16 events of the exact value for values from 512 - 2032 (or 127^4^2), and so on. This represents an event count accuracy of approximately 3%, which is expected to be sufficient for most situations in which a count of events between a start and end event is required.

**Programming Note**

When using the threshold counter, software typically specifies a “threshold counter exceeded n” event in MMCR1. This enables a PMC to count the number of times the counter exceeded a specified threshold value during the time Performance Monitor alerts were enabled.

### 9.4.4 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register as shown below.

<table>
<thead>
<tr>
<th>MMCR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 65. Monitor Mode Control Register 0**

Some bits of MMCR0 are altered by the hardware when various events occur, and some bits may be altered by software.
The following notation is used in the definitions below. When MMCR0PMCC = 11, “PMCs” refers to PMCs 1 - 4 and “PMCj” or PMCjCE refers to “PMCj or PMCjCE, respectively, where j = 2 - 4; otherwise, “PMCs” refers to PMCs 1 - 6 and “PMCj” or PMCjCE refers to “PMCj or PMCjCE, respectively, where j = 2 - 6.

When the MMCR0PMCC is set to 10 or 11, providing problem state programs read/write access, only FC, PMAE, PMAO, can be accessed. All other bits are not changed when mtspr is executed in problem state, and all other bits return 0s when mfspr is executed.

**Programming Note**

When PMCC=10 or 11, problem state programs have access to MMCR0 in order to enable event-based branch routines to reset the FC bit after it has been set to 1 as a result of an enabled condition or event (FCECE=1.) During event processing, the event-based branch handler would write the desired initial values to the PMCs and reset the FC bit to 0. PMAO and PMAE can also be set to their appropriate values during the same write operation before returning.

If additional event-based branches are defined in the future, it may be desirable to set PMAE and PMAO using the BESCRS since additional bits in BESCRS will also need to be written.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td>Freeze Counters (FC)</td>
</tr>
<tr>
<td>0:31</td>
<td>The PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1:31</td>
<td>The PMCs are not incremented.</td>
</tr>
<tr>
<td>33</td>
<td>Freeze Counters and BHRB in Privileged State (FCS)</td>
</tr>
<tr>
<td>0:31</td>
<td>The PMCs are incremented (if permitted by other MMCR bits), and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).</td>
</tr>
<tr>
<td>1:31</td>
<td>The PMCs are not incremented and entries are not written into the BHRB if MSRRHV=0b00.</td>
</tr>
<tr>
<td>34</td>
<td>Conditionally Freeze Counters and BHRB in Problem State (FCP)</td>
</tr>
<tr>
<td>0:31</td>
<td>If the value of bit 51 (FCPC) is 0, this field has the following meaning.</td>
</tr>
<tr>
<td>0:31</td>
<td>The PMCs are incremented (if permitted by other MMCR bits) and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).</td>
</tr>
</tbody>
</table>

**Programming Note**

In order to freeze counters in problem state regardless of MSRHV, MMCR0FCPC must be set to 0 and MMCR0FCP must be set to 1.

**Freeze Counters while Mark = 1 (FCM1)**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Freeze Counters while Mark = 1 (FCM1)</td>
</tr>
<tr>
<td>0:31</td>
<td>The PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1:31</td>
<td>The PMCs are not incremented if MSRPMM=1.</td>
</tr>
</tbody>
</table>

**Freeze Counters while Mark = 0 (FCM0)**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>Freeze Counters while Mark = 0 (FCM0)</td>
</tr>
<tr>
<td>0:31</td>
<td>The PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1:31</td>
<td>The PMCs are not incremented if MSRPMM=0.</td>
</tr>
</tbody>
</table>

**Performance Monitor Alert Enable (PMAE)**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Performance Monitor Alert Enable (PMAE)</td>
</tr>
<tr>
<td>0:31</td>
<td>Performance Monitor alerts are disabled and BHRB entries are not written.</td>
</tr>
<tr>
<td>1:31</td>
<td>Performance Monitor alerts are enabled and BHRB entries are written (if enabled by other bits) until a Performance Monitor alert occurs, at which time:</td>
</tr>
<tr>
<td>0:31</td>
<td>MMCR0PMAE is set to 0.</td>
</tr>
<tr>
<td>1:31</td>
<td>MMCR0PMAO is set to 1.</td>
</tr>
</tbody>
</table>

**Programming Note**

Software can set this bit and MMCR0PMAO to 0 to prevent Performance Monitor exceptions.

Software can set this bit to 1 and then poll the bit to determine whether an enabled condition or event has occurred. This is especially useful for software that runs with MSREE=0.

In earlier versions of the architecture that lacked the concept of Performance Monitor alerts, this bit was called Performance Monitor Exception Enable (PMXE).
38 Freeze Counters on Enabled Condition or Event (FCECE)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when MMCR0TRIGGER=0, at which time:
   ■ MMCR0FC is set to 1
If the enabled condition or event occurs when MMCR0TRIGGER=1, the FCECE bit is treated as if it were 0.

39:40 Time Base Selector (TBSEL)

This field selects the Time Base bit that can cause a Time Base transition event (the event occurs when the selected bit changes from 0 to 1).

00 Time Base bit 63 is selected.
01 Time Base bit 55 is selected.
10 Time Base bit 51 is selected.
11 Time Base bit 47 is selected.

Programming Note

Time Base transition events can be used to collect information about activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multi-threaded systems in which the Time Base registers are synchronized among the threads, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several threads. For this use, software must specify the same TBSEL value for all the threads in the system.

Because the frequency of the Time Base is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.

41 Time Base Event Enable (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

Programming Note

When PMC3 is configured to count the occurrence of Time Base transition events, the events are counted regardless of the value of MMCR0TBEE. (See Section 9.4.5.) The occurrence of a Time Base transition causes a Performance Monitor alert only if MMCR0TBEE=1.

42 BHRB Access (BHRBA)

This field controls whether the BHRB instructions are available in problem state. If an attempt is made to execute a BHRB instruction is made in problem state when the BHRB instructions are not available, a Facility Unavailable interrupt will occur.

0 mfbhrb and clrbhrb are not available in problem state.
1 mfbhrb and clrbhrb are available in problem state unless they have been made unavailable by some other register.

43 Performance Monitor Event-Based Branch Enable (EBE)

This field controls whether Performance Monitor event-based branches are enabled.

When Performance Monitor event-based branches are disabled, no Performance Monitor event-based branches occur regardless of the state of BESCRPME.

0 Performance Monitor event-based branches are disabled.
1 Performance Monitor event-based branches are enabled.

Programming Note

In order to enable a problem state applications to use the Event-Based Branch facility for Performance Monitor events, privileged software initializes MMCR1 to specify the events to be counted, and sets MMCR2, and MMCRA to specify additional sampling controls. MMCR0 should be initialized with PMCC set to 10 or 11 (to give problem state access to various Performance Monitor registers), PMAE and PMAO set to 0s (disabling Performance Monitor alerts), and EBE set to 1 (enabling event-based branches to occur). If the Event-Based Branch facility has not been enabled in the FSCR and HFSCR, it must be enabled in these registers as well.

The above operations by the operating system enable the application to control Performance Monitor event-based branching by means of BESCRPME (to enable or disable Performance Monitor event-based branching) and MMCR0PMAE (to enable or disable Performance Monitor alerts). The requirement to set both BESCRPME and MMCR0PMAE is not expected to impact application performance since MMCR0PMAE as well as other MMCR0 bits that need to be set can be done in a single mtspr instruction.

44:45 PMC Control (PMCC)
This field controls whether or not PMCs 5 - 6 are included in the Performance Monitor, and the accessibility of groups A and B (see Section 9.4.1) of non-privileged SPRs in problem state as described below.

<table>
<thead>
<tr>
<th></th>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PMCs 5 - 6 are included in the Performance Monitor. Group A is read-only in problem state. If an attempt is made to write to an SPR in group A in problem state, a Hypervisor Emulation Assistance interrupt will occur.</td>
</tr>
<tr>
<td>01</td>
<td>PMCs 5 - 6 are included in the Performance Monitor. Group A is not allowed to be read or written in problem state, and group B is not allowed to be read. If an attempt is made to read or write to an SPR in group A, or to read from an SPR in group B in problem state, a Facility Unavailable interrupt will occur.</td>
</tr>
<tr>
<td>10</td>
<td>PMCs 5 - 6 are included in the Performance Monitor. Group A is allowed to be read and written in problem state, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state. If an attempt is made to read MMCR1 in problem state, a Facility Unavailable interrupt will occur.</td>
</tr>
<tr>
<td>11</td>
<td>PMCs 5 - 6 are not included in the Performance Monitor. See Section 9.4.2 for details. Group A except for PMCs 5-6 (SPRs 775,776) is allowed to be read and written, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state. If an attempt is made to read or write to PMCs 5-6 (SPRs 775,776), or to read from MMCR1 in problem state, a Facility Unavailable interrupt will occur. When an SPR is made available by the PMCC field, it is available only if it has not been made unavailable by some other register.</td>
</tr>
</tbody>
</table>

Programming Note

The PMCC field does not affect the behavior of the privileged Performance Monitor Registers (SPRs 784-792, 795-798); accesses to these SPRs in problem state result in Privileged Instruction Type Program interrupts.

The PMCC field also does not affect the behavior of write operations to Group B; write operations to SPRs in Group B are treated as not supported regardless of privilege state. See the mtspr instruction description in Section 4.4.4 for additional information on accessing SPRs that are not supported.

Programming Note

When the PCR makes SPRs unavailable in problem state, they are treated as not implemented, and they are not included in groups A or B regardless of the value of PMCC. Thus when the PCR indicates a version of the architecture prior to V 2.07 (i.e. PCR_{v2.07}=1), the PMCC field does not affect the newly-defined SPRs MMCR2 or SIER; these SPRs are treated as unimplemented registers. Accesses to them in problem state result in Hypervisor Emulation Assistance interrupts regardless of the value of PMCC, and Facility Unavailable interrupts do not occur. See Section 2.6 for additional information.

Freeze Counters in Transactional State (FCTS)

<table>
<thead>
<tr>
<th></th>
<th>Freeze Counters in Non-Transactional State (FCNTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PMCs are incremented (if permitted by other MMCR bits).</td>
</tr>
<tr>
<td>1</td>
<td>PMCs are not incremented when Transactional Memory is in transactional state.</td>
</tr>
</tbody>
</table>
0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when Transactional Memory is in non-transactional state.

48 **PMC1 Condition Enable (PMC1CE)**
This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.
0 Counter negative conditions for PMC1 are disabled.
1 Counter negative conditions for PMC1 are enabled.

49 **PMCj Condition Enable (PMCjCE)**
This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.
0 Counter negative conditions for all PMCjs except those enabled by PMC1CE, are disabled.
1 Counter negative conditions for all PMCjs are enabled.

50 **Trigger (TRIGGER)**
0 The PMCs are incremented (if permitted by other MMCR bits).
1 PMC1 is incremented (if permitted by other MMCR bits). The PMCs are not incremented until PMC1 is negative or an enabled condition or event occurs, at which time:
- the PMCs resume incrementing (if permitted by other MMCR bits)
- MMCR0\_TRIGGER is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

51 **Freeze Counters and BHRB in Problem State Condition (FCPC)**
This bit controls the operation of bit 34 (FCP). See the definition of bit 34 for details.

**Programming Note**
Uses of TRIGGER include the following.
- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMAE=1 (if a Performance Monitor interrupt is desired)

- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMAE=1

52:54 **Performance Monitor Alert Qualifier (PMAOQ)**
These bits provide additional implementation-dependent information about the cause of the Performance Monitor Alert. These bits must be set to 0s when no additional information is available.

55 **Control Counters 5 and 6 With RunLatch (CC56RUN)**
When MMCR0_PMCC = b'11', the setting of this bit has no effect; otherwise it is defined as follows.

0  PMCs 5 and 6 are incremented if CTRL_RUN=1 if permitted by other MMCR bits.
1  PMCs 5 and 6 are incremented, regardless of the value of CTRL_RUN, if permitted by other MMCR bits.

0 The PMCs are incremented (if permitted by other MMCR bits) and BHRB entries are written (if permitted by the BHRB Instruction Filtering Mode field in MMCRa).
1  The PMCs are not incremented and BHRB entries are not written if MSR_HV_PR=0b10.

9.4.5 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register as shown below.

```
<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Problem state access (SPR 782)</td>
</tr>
<tr>
<td>5:6</td>
<td>Privileged access (SPR 782 or 798)</td>
</tr>
<tr>
<td>13</td>
<td>Implementation-dependent</td>
</tr>
</tbody>
</table>
```

Software can set this bit to 1 and set PMAE to 0 to simulate the occurrence of a Performance Monitor alert.

Software should set this bit to 0 after handling the Performance Monitor alert.
### PMC1 Selector (PMC1SEL)

The value of PMC1SEL specifies the event to be counted by PMC 1 as defined below. All values in the range of C0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA<sub>SE</sub>=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA<sub>SM</sub>.)

- **E0**: The thread has dispatched a randomly sampled instruction. (RIS)
- **E2**: The thread has completed a randomly sampled Branch instruction for which the branch was taken. (RIS, RBS)
- **E4**: The thread has failed to locate a randomly sampled instruction in the primary cache. (RIS)
- **E6**: The threshold event counter has exceeded the number of events corresponding to threshold A (see Table 3). (RIS, RLS, RBS)
- **E8**: The threshold event counter has exceeded the number of events corresponding to threshold E (see Table 3). (RIS, RLS, RBS)
- **EA**: The thread filled a block in a cache containing data accessed by a randomly sampled Load instruction. (RIS, RLS)
- **EC**: The threshold event counter has reached its maximum value when random sampling is enabled. (RIS, RLS, RBS)
- **F0**: A cycle has occurred. This event is not controlled by MMCRC<sub>FC1</sub><sub>WAIT</sub>.
- **F2**: A cycle has occurred in which the thread completed one or more instructions.
- **F4**: The thread has completed a Floating Point, Vector Floating Point, or VSX Floating Point instruction other than a load or store operation up to the point at which it has reported all exceptions that it will cause.
- **F6**: The thread has failed to locate an ERAT entry during instruction address translation.
- **F8**: A cycle has occurred in which all previously initiated instructions have completed and no instructions are available for initiation.
- **FA**: A cycle has occurred during which the RUN bit of the control register for one or more threads was set to 1.

### PMC2 Selector (PMC2SEL)

The value of PMC2SEL specifies the event to be counted by PMC 2 as defined below. All values in the range of C0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA<sub>SE</sub>=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA<sub>SM</sub>.)

- **E0**: The thread located the data for a randomly sampled Load instruction from storage that did not reside in any cache. (RIS, RLS)
- **E2**: The thread has failed to locate the data for a randomly sampled Load instruction in the primary data cache. (RIS, RLS)
- **E4**: The thread filled a block in the primary cache containing data accessed by a randomly sampled Load instruction from a location other than the secondary or tertiary cache. (RIS, RLS)
- **E6**: The threshold event counter has exceeded the number of events corresponding to threshold B (see Table 3). (RIS, RLS, RBS)
- **E8**: The threshold event counter has exceeded the number of events corresponding to threshold F (see Table 3). (RIS, RLS, RBS)
- **F0**: A cycle has occurred during which the thread's CTRL register was set to 1.
- **F2**: The thread has dispatched an instruction.
- **F4**: A cycle has occurred during which the thread has completed a Branch instruction for which the branch was taken.
- **F6**: The thread has failed to locate an ERAT entry during data address translation, and a new ERAT entry corresponding to the data address has been written.
- **F8**: An external interrupt for the thread has occurred.
- **FA**: The thread has completed a Branch instruction in the primary cache.
- **FC**: The thread has failed to locate an instruction in the primary cache.
- **FE**: The thread has filled a block in the primary cache containing data accessed by a Load instruction from a location other than the secondary cache.
### PMC3 Selector (PMC3SEL)

The value of PMC3SEL specifies the event to be counted by PMC 3 as defined below. All values in the range of C0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA$_{SE}=1$). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA$_{SM}$.)

- **E2** The thread has completed a randomly sampled *Store* instruction up to the point at which it has reported all exceptions that it will cause. (RIS)
- **E4** The thread has incorrectly predicted the outcome of a randomly sampled *Branch* instruction. (RIS, RBS)
- **E6** The thread has failed to locate a data ERAT entry for a randomly sampled instruction. (RIS)
- **E8** The threshold event counter has exceeded the number of events corresponding to threshold C (see Table 3). (RIS, RLS, RBS)
- **EA** The threshold event counter has exceeded the number of events corresponding to threshold G (see Table 3). (RIS, RLS, RBS)
- **F0** The thread has attempted to store data in the primary cache but the block containing the effective address did not exist.
- **F2** The thread has dispatched an instruction.
- **F4** The thread has completed an instruction during a time when the RUN bit of the CTRL register for all enabled threads on the multi-threaded processor is set to 1.
- **F6** The thread has filled a block in the primary cache with data accessed by a *Load* instruction.
- **F8** A Time Base transition event has occurred for the thread. This event is counted regardless of whether or not Time Base transition events are enabled by MMCRO$_{TBE}$.
- **FA** The thread has loaded an instruction from a higher level cache than the tertiary cache.
- **FC** A thread was unable to translate an effective data address using the TLB.
- **FE** The thread has filled a block in the primary cache containing data accessed by a *Load* instruction with data from a location other than the secondary or tertiary cache.

### PMC4 Selector (PMC4SEL)

The value of PMC4SEL specifies the event to be counted by PMC 4 as defined below. All values in the range of C0 - FF that are not specified below are reserved.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Disable events. (No events occur.)</td>
</tr>
<tr>
<td>01-BF</td>
<td>Implementation dependent</td>
</tr>
<tr>
<td>C0-DF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The following events can occur only when random sampling is enabled (MMCRA$_{SE}=1$). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in MMCRA$_{SM}$.)

- **E0** The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
- **E4** A thread was unable to translate an effective data address using the TLB. (RIS)
- **E6** The thread has loaded a randomly sampled instruction from a location at a higher level than the tertiary cache. (RIS)
- **E8** The thread filled a block in the primary cache containing data accessed by a randomly sampled *Load* instruction from a location other than the secondary cache. (RIS, RLS)
- **EA** The threshold event counter has exceeded the number of events corresponding to threshold D (see Table 3). (RIS, RLS, RBS)
- **EC** The threshold event counter has exceeded the number of events corresponding to threshold H (see Table 3). (RIS, RLS, RBS)
- **F0** The thread has attempted to load data from the primary cache but the block containing the effective address did not exist.
- **F2** A cycle has occurred in which the thread has dispatched one or more instructions.
- **F4** A cycle has occurred in which the PURR was incremented when the RUN bit of the CTRL register was set to 1.
- **F6** The thread has incorrectly predicted the outcome of a branch instruction.
- **F8** The thread has discarded fetched instructions.
- **FA** The thread has completed an instruction when the RUN bit of the CTRL register is set to 1.
- **FC** A thread was unable to translate an effective instruction address using the TLB and has stored an entry corresponding to the instructions effective address in the TLB.
- **FE** The thread has loaded data for a *Load* instruction from storage that did not reside in any cache.
9.4.6 Monitor Mode Control Register 2

Monitor Mode Control Register 2 (MMCR2) is a 64-bit register that contains 9-bit control fields for controlling the operation of PMC1 - PMC6 as shown below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>0</th>
<th>8</th>
<th>9</th>
<th>17</th>
<th>18</th>
<th>25</th>
<th>26</th>
<th>35</th>
<th>36</th>
<th>44</th>
<th>45</th>
<th>53</th>
<th>54</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Freeze Counter n in Privileged State (FCnS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if MSR_HV_PR=0b00.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Freeze Counter n in Problem State if MSR_HV=0 (FCnP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if MSR_HV_PR=0b01.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 67. Monitor Mode Control Register 2

When MMCR0_PMCC = b’11’, fields C1 - C4 control the operation of PMC1 - PMC4, respectively and fields C5 and C6 are meaningless; otherwise, fields C1 - C6 control the operation of PMC1 - PMC6, respectively. The bit definitions of each Cn field are as follows, where n = 1,...,6.

When the MMCR0_PMCC is set to 10 or 11, providing problem state programs read/write access, only the FCnP bits can be accessed. All other bits are not changed when *mtspr* is executed in problem state, and all other bits return 0s when *mfspr* is executed.

Bits 54:63 of MMCR2 are implementation dependent.

9.4.7 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register as defined below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>0</th>
<th>8</th>
<th>9</th>
<th>17</th>
<th>18</th>
<th>25</th>
<th>26</th>
<th>35</th>
<th>36</th>
<th>44</th>
<th>45</th>
<th>53</th>
<th>54</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Freeze Counter n in Problem State if MSR_HV=1 (FCnP1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if MSR_HV_PR=0b11.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Freeze Counter n while Mark = 1 (FCnM1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if MSR_PMM=1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Freeze Counter n while Mark = 0 (FCnM0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if MSR_PMM=0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Freeze Counter n in Wait State (FCnWAIT)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if CTRL_RUN=0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Software is expected to set CTRL_RUN=0 when it is in a &quot;wait state&quot;, i.e, when there is no process ready to run.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Freeze Counter n in Hypervisor State (FCnH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 PMCn is incremented (if permitted by other MMCR bits).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 PMCn is not incremented if MSR_HV_PR=0b10.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits 54:63 of MMCR2 are implementation dependent.
read-only access, all fields can be read. When MMCR0.PMCC is set to 10 or 11, the Threshold Event Counter Exponent (TECX) and Threshold Event Counter Mantissa (TECM) fields are read-only, and all other fields return 0s when mfspr is executed; all fields are not changed when mspr is executed in problem state.

The bit definitions of MMCRA are as follows.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Problem state access (SPR 770)</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>32:33</td>
<td>BHRB Instruction Filtering Mode (IFM)</td>
</tr>
<tr>
<td></td>
<td>This field controls the filter criterion used by</td>
</tr>
<tr>
<td></td>
<td>the hardware when recording Branch instructions</td>
</tr>
<tr>
<td></td>
<td>in the BHRB. See Section 9.5.</td>
</tr>
<tr>
<td>00</td>
<td>No filtering</td>
</tr>
<tr>
<td>01</td>
<td>Do not record any Branch instructions unless the</td>
</tr>
<tr>
<td></td>
<td>LK field is set to 1.</td>
</tr>
<tr>
<td>10</td>
<td>Do not record I-Form instructions. For B-Form and</td>
</tr>
<tr>
<td></td>
<td>XL-Form instructions for which the BO field</td>
</tr>
<tr>
<td></td>
<td>indicates &quot;Branch always,&quot; do not record the</td>
</tr>
<tr>
<td></td>
<td>instruction if it is B-Form and do not record</td>
</tr>
<tr>
<td></td>
<td>the instruction address but record only the</td>
</tr>
<tr>
<td></td>
<td>branch target address if it is XL-Form.</td>
</tr>
<tr>
<td>11</td>
<td>Filter and enter BHRB entries as for mode 10,</td>
</tr>
<tr>
<td></td>
<td>but for B-Form and XL-Form instructions for which</td>
</tr>
<tr>
<td></td>
<td>BO[0]=1 or for which the “a” bit in the BO field</td>
</tr>
<tr>
<td></td>
<td>is set to 1, do not record the instruction if it</td>
</tr>
<tr>
<td></td>
<td>is B-Form and do not record the instruction</td>
</tr>
<tr>
<td></td>
<td>address but record only the branch target</td>
</tr>
<tr>
<td></td>
<td>address if it is XL-Form.</td>
</tr>
</tbody>
</table>

**Programming Note**

The filters provided by the 10 and 11 values of IFM field can be restated in terms of the operation performed as follows:

00 Do not record the instruction address of any unconditional Branch instruction; record only the target address of XL-form unconditional Branch instructions.

10 Filter as for encoding 10, but for conditional Branch instructions that provide a hint or that do not depend on the value of CRBI, do not record the instruction if it is B-Form and record only the target address if it is XL-Form.

34:36 **Threshold Event Counter Exponent (TECX)**

This field species the exponent of the threshold event counter value. See Section 9.4.3 for additional information. The maximum exponent supported must be at least 5.

37 Reserved

38:44 **Threshold Event Counter Mantissa (TECM)**

This field species the mantissa of the threshold event counter value. See Section 9.4.3 for additional information.

**Programming Note**

Problem state programs need to read the TECX and TECM fields during processing of an event-based branch that occurs due to a “threshold counter exceeded n” event. All the other fields will typically be initialized by the operating system.

45:47 **Threshold Event Counter Event (TECE)**

This field specifies that is counted by the Threshold Event Counter. The values and meanings are specified as follows.

<table>
<thead>
<tr>
<th>Value</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Disable Counting</td>
</tr>
<tr>
<td>001</td>
<td>A cycle has occurred</td>
</tr>
<tr>
<td>010</td>
<td>An instruction has completed.</td>
</tr>
<tr>
<td>011</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

All other values are implementation-dependent.

48:51 **Threshold Start Event (TS)**

This field specifies the event that causes the threshold event counter to start counting cycles. The events only occur if MMCRA[SE]=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode is specified in MMCRA[SM].) 0000 Reserved.

0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)

0010 The thread has dispatched a randomly sampled instruction. (RIS)

0011 A randomly sampled instruction has been sent to a facility (e.g. Branch, Fixed Point, etc.) (RIS, RLS, RBS)

0100 The thread has completed a randomly sampled instruction up to the point at which it has caused all the exceptions that it will cause. (RIS, RLS, RBS)

0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)

0110 The thread has failed to locate data for a randomly sampled Load instruction in the primary data cache. (RIS, RLS)
0111 The thread has filled a block in the primary cache containing data accessed by a randomly sampled Load instruction. (RIS, RLS)

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

Threshold End Event (TE)

This field specifies the event that causes the threshold event counter to stop counting cycles. The events only occur if MMCRA\textsubscript{SE}=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode is specified in MMCRA\textsubscript{SM}.)

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

Eligibility for Random Sampling (ES)

When random sampling is enabled (SE=1) and the SM field indicates random instruction sampling (RIS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 All instructions
001 All Load and Store instructions
010 All probe no-op instructions
011 Reserved.

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

When random sampling is enabled (SE=1) and the SM field indicates random load/store facility sampling (RLS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 Instructions for which the thread has attempted to load data from the cache but the block containing the effective address did not exist.
001 Reserved
010 Reserved
011 Reserved

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

When random sampling is enabled (ES=1) and the SM field indicates random Branch Facility sampling (RBS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 Instructions for which the thread has incorrectly predicted the outcome of a Branch instruction.
001 Instructions for which the thread has incorrectly predicted the outcome of a Branch instruction because the Condition Register was different from the predicted value.
010 Instructions for which the thread has incorrectly predicted the target address of a Branch instruction.
011 All Branch instructions for which the branch was taken.

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
100 - 111 - Reserved
Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

60 Reserved

61:62 Random Sampling Mode (SM)

00 Random Instruction Sampling (RIS) - Instructions that meet the criterion specified in the ES field for random instruction sampling are eligible to be sampled.

01 Random Load/Store Facility Sampling (RLS) - Instructions that meet the criterion specified in the ES field for random Load/Store Facility sampling are eligible for sampling.

10 Random Branch Facility Sampling (RBS) Instructions that meet the criterion specified in the ES field for random Branch Facility sampling are eligible for sampling.

11 Reserved

63 Random Sample Enable (SE)

0 Continuous sampling is enabled.
1 Random sampling is enabled.

See Section 9.4.8 for information about sampling.

9.4.8 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register.

| SIAR | 0 | 63 |

Figure 69. Sampled Instruction Address Register

When a Performance Monitor alert occurs because of an event that occurred due to a randomly sampled instruction, the SIAR contains the effective address of the instruction if SIER_{SIARV} = 1 and is invalid if SIER_{SIARV} = 0.

When a Performance Monitor alert occurs because of an event other than an event that occurred due to a randomly sampled instruction, the SIAR contains the effective address of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred.

The contents of SIAR may be altered by the hardware if and only if MMCR0_{PMAE} = 1. Thus after the Performance Monitor alert occurs, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, the value of MSR_{HV PR} that was in effect when the sampled instruction was being executed is reported in SIER.

9.4.9 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register.

| SDAR | 0 | 63 |

Figure 70. Sampled Data Address Register

When a Performance Monitor alert occurs because of an event that occurred due to a randomly sampled instruction, the SDAR contains the effective address of the data accessed by the instruction if SIER_{SDARV} = 1 and is invalid if SIER_{SDARV} = 0.

When a Performance Monitor alert occurs because of an event other than an event that occurred due to a randomly sampled instruction, the SDAR contains the effective address of the storage operand of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This storage operand may or may not be the storage operand (if any) associated with the instruction whose address is in the SIAR.

The contents of SDAR may be altered by the hardware if and only if MMCR0_{PMAE} = 1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the hardware until software sets MMCR0_{PMAE} to 1. After software sets MMCR0_{PMAE} to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, MMCRA indicates whether the sampled data is the storage operand of the sampled instruction.

9.4.10 Sampled Instruction Event Register

The Sampled Instruction Event Register (SIER) is a 64-bit register.

When random sampling is enabled and a Performance Monitor Alert occurs because of an event caused by execution of a randomly sampled instruction, this register contains information about the sampled instruction.
(i.e. the instruction whose effective address is contained in the SIAR) when a Performance Monitor alert occurred. All fields are valid unless otherwise indicated.

**Programming Note**

A Performance Monitor Alert is caused by execution of a randomly sampled instruction if any of the following apply when the Performance Monitor Alert occurs:

- random sampling was enabled and a counter negative condition exists in a PMC that was counting events based on randomly sampled instructions
- a threshold start event was specified in MMCRA and the threshold event counter has reached its maximum value.

When random sampling is disabled or when a Performance Monitor Alert occurs because of an event that was not caused by execution of a randomly sampled instruction, then the contents of this register are meaningless.

**Figure 71. Sampled Instruction Event Register**

The contents of SIER may be altered by the hardware if and only if MMCR0\textsubscript{PMAE}=1. Thus after the Performance Monitor alert occurs, the contents of SIER are not altered by the hardware until software sets MMCR0\textsubscript{PMAE} to 1. After software sets MMCR0\textsubscript{PMAE} to 1, the contents of SIER are undefined until the next Performance Monitor alert occurs.

The bit definitions of the SIER are as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:37</td>
<td>These bits have different definitions for problem state than for other privilege states.</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Sampled MSR\textsubscript{PP} (SAMPPR)</td>
<td>Value of MSR\textsubscript{PP} when the Performance Monitor Alert occurred.</td>
</tr>
<tr>
<td>39</td>
<td>Sampled MSR\textsubscript{HV} (SAMPHV)</td>
<td>Value of MSR\textsubscript{HV} when the Performance Monitor Alert occurred.</td>
</tr>
<tr>
<td>40</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>SIAR Valid (SIARV)</td>
<td>Set to 1 when the contents of the SIAR are valid (i.e. they contain the effective address of the sampled instruction); otherwise set to 0.</td>
</tr>
<tr>
<td>42</td>
<td>SDAR Valid (SDARV)</td>
<td>Set to 1 when the contents of the SDAR are valid (i.e. they contain the effective address of the data accessed by the sampled instruction); otherwise set to 0.</td>
</tr>
<tr>
<td>43</td>
<td>Threshold Exceeded (TE)</td>
<td>Set to 1 by the hardware if the contents of threshold event counter exceeded the maximum value when the Performance Monitor Alert occurred; otherwise set to 0 by hardware.</td>
</tr>
<tr>
<td>44</td>
<td>Slew Down</td>
<td>Set to 1 by the hardware if the processor clock was lower than nominal when the performance monitor alert occurred; otherwise set to 0 by hardware.</td>
</tr>
<tr>
<td>45</td>
<td>Slew Up</td>
<td>Set to 1 by the hardware if the processor clock was higher than nominal when the performance monitor alert occurred; otherwise set to 0 by hardware.</td>
</tr>
<tr>
<td>46:48</td>
<td>Sampled Instruction Type (SITYPE)</td>
<td>This field indicates the sampled instruction type. The values and their meanings are as follows.</td>
</tr>
<tr>
<td>000</td>
<td>The hardware is unable to indicate the sampled instruction type</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>Load Instruction</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Store instruction</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>Branch Instruction</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Floating Point Instruction other than a Load or Store instruction</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Fixed Point Instruction other than a Load or Store instruction</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Condition Register or System Call Instruction</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>49:51</td>
<td>Sampled Instruction Cache Information (SICACHE)</td>
<td>This field provides cache-related information about the instruction.</td>
</tr>
<tr>
<td>000</td>
<td>The hardware is unable to indicate any cache information for the sampled instruction.</td>
<td></td>
</tr>
</tbody>
</table>
001 The thread located the instruction in the primary cache.
010 The thread located the instruction in the secondary cache.
011 The thread located the instruction in the tertiary cache
100 The thread failed to locate the instruction in the primary, secondary, or tertiary cache
101 Reserved.
110 Reserved
111 Reserved

52 **Sampled Instruction Taken Branch** (SITAKBR)
Set to 1 if the SITYPE field indicates a Branch instruction and the branch was taken; otherwise set to 0.

53 **Sampled Instruction Mispredicted Branch** (SIMISPED)
Set to 1 if the SITYPE field indicates a Branch instruction and the thread incorrectly predicted the outcome of the instruction; otherwise set to 0.

54:55 **Sampled Branch Instruction Misprediction Information** (SIMISPREDICT)
If SIMISPRED=1, this field indicates how the thread incorrectly predicted the outcome of a Branch instruction; otherwise this field is set to 0.

56 **Sampled Instruction Data ERAT Miss** (SIDERAT)
When the SITYPE field indicates a Load or Store instruction, this field is set to 1 if the thread has failed to locate an ERAT entry during data address translation for the sampled instruction and otherwise is set to 0.

When the SITYPE field does not indicate a Load or Store instruction, this field is not valid.

57:59 **Sampled Instruction Data Address Translation Information** (SIDAXLATE)
This field contains information about data address translation for the sampled instruction. The values and their meanings are as follows.
000 The instruction did not access data.
001 The thread was able to translate the effective data address using the TLB.
010 A PTEG entry required for data translation for the instruction was loaded from the secondary cache.
011 A PTEG entry required for data translation for the instruction was loaded from the tertiary cache.
100 A PTEG entry required for data translation for the instruction was loaded from storage that did not reside in any cache.
101 A PTEG entry required for data translation for the instruction was loaded from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110 A PTEG entry required for data translation for the instruction was loaded from a cache on a different chip from the thread.
111 Reserved

60:62 **Sampled Instruction Storage Access Information** (SISAI)
This field contains information about storage accesses made by the sampled instruction. The values and their meanings are as follows.
000 The instruction did not access data.
001 The thread was able to access the data referenced by the instruction from the primary cache.
010 The thread was able to access the data referenced by the instruction from the secondary cache.
011 The thread was able to access the data referenced by the instruction from the tertiary cache.
100 The thread was able to access the data referenced by the instruction from storage that did not reside in any cache.
101 The thread was able to access the data referenced by the instruction from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110 The thread was able to access the data referenced by the instruction from a cache on a different chip from the thread.
111 The instruction was a Store for which the data was placed into a location other than the primary cache.

63 **Sampled Instruction Completed** (SICMPL)
Set to 1 if the sampled instruction has completed; otherwise set to 0.
9.5 Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is described in Book I but only at the level required by application programmers. Additional aspects of the BHRB are described here.

In order to enable problem state programs to use the BHRB, MMCR0BHRBA must be set to 1 to enable execution of \texttt{mfbhrb} and \texttt{clrbhrb} instructions in problem state. Additionally, MMCR0EBE must also be set to 1 to enable Performance Monitor event-based branches, and MMCR0PMCC must be set to 10 or 11 to allow problem state programs to access the necessary Performance Monitor registers. (See Section 9.4.4.)

The BHRB is written by the hardware if and only if performance monitor alerts are enabled by setting MMCR0PMAE to 1. After MMCR0PMAE has been set to 1 and a Performance Monitor alert occurs, MMCR0PMAE is set to 0 and the BHRB is not altered by hardware until software sets MMCR0PMAE to 1 again.

When MMCR0PMAE=1, \texttt{mfbhrbe} instructions return 0s to the target register.

BHRB Entries

When the BHRB is written by hardware, only those Branch instructions that meet the filtering criterion indicated by MMCRAIFM and for which the branch was taken are included.

In addition to the mode of BHRB operation described in Book I, an implementation is allowed to provide an implementation-dependent mode of the operation in which the entries that are written are implementation dependent. If such a mode is provided, the means by which the BHRB is put into this mode must be a hypervisor resource.

9.6 Interaction With Other Facilities

9.6.1 Trace Facility

If the Trace facility includes setting SIAR and SDAR (see Section 6.5.15, “Trace Interrupt [Category: Trace]”), and tracing is active (MSRSE=1 or MSRBE=1), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when MMCR0PMAE=0.

---

**Programming Note**

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.
Chapter 10. External Control [Category: External Control]

The External Control facility permits a program to communicate with a special-purpose device. The facility consists of a Special Purpose Register, called EAR, and two instructions, called External Control In Word Indexed (eciwx) and External Control Out Word Indexed (ecowx).

This facility must provide a means of synchronizing the devices with the hardware to prevent the use of an address by the device when the translation that produced that address is being invalidated.

10.1 External Access Register

This 32-bit Special Purpose Register controls access to the External Control facility and, for external control operations that are permitted, identifies the target device.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>E</td>
<td>Enable bit</td>
</tr>
<tr>
<td>58:63</td>
<td>RID</td>
<td>Resource ID</td>
</tr>
</tbody>
</table>

All other fields are reserved.

Figure 72. External Access Register

The External Access Register (EAR) is a hypervisor resource; see Chapter 2.

The high-order bits of the RID field that correspond to bits of the Resource ID beyond the width of the Resource ID supported by the implementation are treated as reserved bits.

Programming Note

The hypervisor can use the EAR to control which programs are allowed to execute External Access instructions, when they are allowed to do so, and which devices they are allowed to communicate with using these instructions.

10.2 External Access Instructions

The External Access instructions, External Control In Word Indexed (eciwx) and External Control Out Word Indexed (ecowx), are described in Book II. Additional information about them is given below.

If attempt is made to execute either of these instructions when EAR_E=0, a Data Storage interrupt occurs with bit 43 of the DSISR set to 1.

The instructions are supported whenever MSR_DR=1. If either instruction is executed when MSR_DR=0 (real addressing mode), the results are boundedly undefined.
Chapter 11. Processor Control [Category: Server.Processor Control]

11.1 Overview

The Processor Control facility provides a mechanism for the hypervisor to send messages to other threads that are on the same multi-threaded processor. Privileged programs are able to send messages to other threads on the same multi-threaded processor; if the processor is configured into sub-processors, privileged programs can only send messages to other threads on the same sub-processor.

11.2 Programming Model

Both hypervisor-level and privileged-level messages can be sent. Hypervisor-level messages are sent using the `msgsnd` instruction and cause hypervisor-level exceptions when accepted. Privileged-level messages are sent using the `msgsndp` instruction and cause privileged-level exceptions when accepted. For both instructions, the message type is specified in a general purpose register.

11.2.1 Message Type

The message type is specified by the contents of bits 32:36 in the RB operand of the `msgsnd` or `msgsndp` instruction as follows.

Message type for `msgsnd`

<table>
<thead>
<tr>
<th>RB32:36</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Directed Hypervisor Doorbell Interrupt (DH_DBELL)</td>
</tr>
</tbody>
</table>

A Directed Hypervisor Doorbell exception is generated on a thread only after it has filtered and determined that it should accept the message, and the thread is on the same multi-threaded processor as the thread executing the `msgsnd` instruction.

All other values of RB32:36 are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.

Message type for `msgsndp`

<table>
<thead>
<tr>
<th>RB32:36</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Directed Privileged Doorbell Interrupt (DP_DBELL)</td>
</tr>
</tbody>
</table>

A Directed Privileged Doorbell exception is generated on a thread only after it has filtered and determined that it should accept the message, and the following are satisfied:

- for processors not partitioned into sub-processors, the thread is on the same multi-threaded processor as the thread executing the `msgsndp` instruction,
- for processors partitioned into sub-processors, the thread is on the same sub-processor as the thread executing the `msgsndp` instruction.

All other values of RB32:36 are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.

11.2.2 Doorbell Message Payload and Filtering

The message payload is specified by the contents of bits 37:63 in the RB operand of the `msgsnd` or `msgsndp` instruction.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Reserved</td>
</tr>
<tr>
<td>38:56</td>
<td>Reserved</td>
</tr>
<tr>
<td>57:63</td>
<td>TIR Tag (TIRTAG)</td>
</tr>
</tbody>
</table>

For `msgsnd` instructions, the recipient of the message compares this field during the filtering process with its privileged thread number. For `msgsndp` instructions, the recipient of the message compares this field during the filtering process with its hypervisor thread number.
Filtering

The examination of the message payload for the purpose of determining if the message is to be accepted is referred to as filtering.

If a Directed Hypervisor Doorbell message is received by a thread, the message is accepted and the corresponding exception is generated only if the TIRTAG field of the message payload is equal to the hypervisor thread number of the recipient.

If a Directed Privileged Doorbell message is received by a thread, the message is accepted and the corresponding exception is generated only if the TIRTAG field of the message payload is equal to the privileged thread number of the recipient.

If the message is to be accepted, the exception specified by the type field is generated, otherwise the message is ignored. When the exception is generated, the corresponding interrupt occurs when no higher priority exception exists and the interrupt is enabled (MSREEE=1 for the Directed Privileged Doorbell interrupt and MSRE=1 or MSR HV=0 for the Directed Hypervisor Doorbell interrupt).

A Directed Privileged Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a mtsp or msgclrp instruction.

A Directed Hypervisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a mtsp or msgclrp instruction.

If a doorbell exception is present and the corresponding interrupt is pended because MSREEE=0, additional doorbell exceptions are ignored until the exception is cleared.

Programming Note

If msgsndp is executed with TIRTAG set to a value greater than the highest privileged thread number on the sub-processor (or on the processor if sub-processors are not supported), then this instruction behaves as a no-op because the filtering process (see below) prevents the receiving threads from accepting it. Similarly, if msgsnd is executed with TIRTAG set to a value greater than the highest hypervisor thread number on the processor, then the instruction also behaves as a no-op.

11.3 Processor Control Registers

11.3.1 Directed Privileged Doorbell Exception State

The layout of the Directed Privileged Doorbell Exception State (DPDES) register is shown in below.

<table>
<thead>
<tr>
<th>DPDES</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Figure 73. Directed Privileged Doorbell Exception State Register

The DPDES register is a 64-bit register. For t < T, where T is the number of threads on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), bit 63-t corresponds to the thread with privileged thread number t.

When the contents of DPDES_63-t change from 0 to 1, a Directed Privileged Doorbell exception will come into existence within a reasonable period of time. When the contents of DPDES_63-t change from 1 to 0, the existing Directed Privileged Doorbell exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DPDES_63-t is the result a msgsndp or msgclrp instruction or of modification of the DPDES register caused by execution of an mtsp instruction.

Bits 0:63-T of the DPDES are reserved.

11.3.2 Directed Hypervisor Doorbell Exception State

The layout of the Directed Hypervisor Doorbell Exception State (DPDES) register is shown in below.

<table>
<thead>
<tr>
<th>DHDES</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Figure 74. Directed Hypervisor Doorbell Exception State Register

The DHDES register is a 64-bit register. For t < T, where T is the number of threads on the multi-threaded processor, bit 63-t corresponds to the thread with hypervisor thread number t.

When the contents of DHDES_63-t change from 0 to 1, a Directed Hypervisor Doorbell exception will come into existence within a reasonable period of time. When the contents of DHDES_63-t change from 1 to 0, the existing Directed Hypervisor Doorbell exception, if any, will
cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DHDES₆₃₋₁ is the result a \texttt{msgsnd} or \texttt{msgclr} instruction or of modification of the DHDES register caused by execution of an \texttt{mtspr} instruction.

Bits 0:63-T of the DHDES are reserved.
11.4 Processor Control Instructions

`msgsnd`, `msgsndp`, `msgclr`, and `msgclrp` instructions are provided for sending and clearing messages. `msgsnd` and `msgsndp` are privileged instructions, while `msgclr` and `msgclrp` are hypervisor privileged instructions.

### Message Send

#### X-form

```
msgsnd   RB
31 /// /// RB 206 /
```

- `msgtype` ← GPR(RB)_{32:36}
- `payload` ← GPR(RB)_{37:63}
- `t` ← RB_{57:63}
- IF(`msgtype` = 0x05) and `t` ≤ maximum hypervisor thread number on processor
  then
  DHDES_{63-63-t} ← 1
  send_msg(`msgtype`, `payload`)

`msgsnd` sends a message to other threads.

Let `msgtype` be (RB)_{32:36}, let message `payload` be (RB)_{37:63}, and let `t` be the hypervisor thread number of the thread executing the `msgsnd`. If `msgtype` = 0x05 and `t` is less than or equal to the maximum hypervisor thread number on the same multithreaded processor (or sub-processor if sub-processors are supported), then send the Directed Hypervisor Doorbell message to thread `t` on the same multithreaded processor and set DPDES_{63-63-t} to 1.

The actions taken on receipt of a message are defined in Section 11.2.2.

This instruction is hypervisor privileged.

**Special Registers Altered:**

- DHDES

---

**Programming Note**

If `msgsnd` is used to notify the receiver that updates have been made to storage, a `sync` should be placed between the stores and the `msgsnd`. See Section 5.9.2.

### Message Clear

#### X-form

```
msgclr   RB
31 /// /// RB 238 /
```

- `msgtype` ← (RB)_{32:36}
- IF(`msgtype` = 0x05)
  then
  `t` ← hypervisor thread number of executing thread
  DHDES_{63-t} ← 0

`msgclr` clears a message of `msgtype` previously accepted by the thread executing the `msgclr`.

Let `msgtype` be (RB)_{32:36}, and let `t` be the hypervisor thread number of the thread executing the `msgclr`. If `msgtype` = 0x05, then clear any exception that exists for this message type by setting DHDES_{63-t} to 0; otherwise do not modify DHDES or clear any exceptions for this message.

Execution of `msgclr` ensures that the formerly pending exception will not result in an interrupt when the corresponding interrupt class is reenabled.

The types of messages that can be cleared are defined in Section 11.2.1.

This instruction is hypervisor privileged.

**Special Registers Altered:**

- DHDES

---

**Programming Note**

`msgclr` is typically issued only when MSR_{EE}=0. If `msgclr` is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.
**Message Send Privileged X-form**

<table>
<thead>
<tr>
<th>msgsndp</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB</td>
<td>31</td>
</tr>
</tbody>
</table>

msgtype ← GPR(RB)_{32:36}
payload ← GPR(RB)_{37:63}
t ← RB_{57:63}
IF(msgtype = 0x05) and
  t ≤ maximum privileged thread number
  on processor or sub-processor
  then
  DPDES_{63-t} ← 1
  send_msg(msgtype, payload)

**msgsndp** sends a message to other threads.

Let msgtype be (RB)_{32:36}, let message payload be (RB_{37:63}), and let t be the privileged thread number indicated in RB_{57:63}. If msgtype = 0x05 and t is less than or equal to the maximum privileged thread number on the multi-threaded processor (or on the sub-processor if sub-processors are supported), then send the Directed Privileged Doorbell message to thread t on the same multithreaded processor (or sub-processor if sub-processors are supported), and set DPDES_{63-t} to 1.

The actions taken on receipt of a message are defined in Section 11.2.2.

This instruction is privileged.

**Special Registers Altered:**

DPDES

---

**Programming Note**

If **msgsndp** is used to notify the receiver that updates have been made to storage, a sync should be placed between the stores and the **msgsnd**. See Section 5.9.2.

**Message Clear Privileged X-form**

<table>
<thead>
<tr>
<th>msgclrp</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB</td>
<td>31</td>
</tr>
</tbody>
</table>

msgtype ← (RB)_{32:36}
IF(msgtype = 0x05)
  then
  t ← privileged thread number of executing thread
  DPDES_{63-t} ← 0

**msgclrp** clears a message of msgtype previously accepted by the thread executing the **msgclrp**.

Let msgtype be (RB)_{32:36}, and let t be the thread number of the thread executing the **msgclrp**. If msgtype = 0x05, then clear any exception that exists for this message type by setting DPDES_{63-t} to 0; otherwise do not modify DPDES or clear any exceptions for this message.

Execution of **msgclrp** ensures that the formerly pending exception will not result in an interrupt when the corresponding interrupt class is reenabled.

The types of messages that can be cleared are defined in Section 11.2.1.

This instruction is privileged.

**Special Registers Altered:**

DPDES

---

**Programming Note**

**msgclrp** is typically issued only when MSR_{EE}=0. If **msgclrp** is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.
Chapter 12. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of SLB entries, or the contents of other system resources that control the context in which a program executes can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing MSRIR from 0 to 1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a context-altering instruction. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 4 (for data access) and Table 5 (for instruction fetch and execution).

The notation “CSI” in the tables means any context synchronizing instruction (e.g., sc, isync, or rfid). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like “the synchronizing instruction”, below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, “the synchronizing instruction before (after) the context-altering instruction” should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

**Programming Note**

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as the rfid that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 5, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.

---

Chapter 12. Synchronization Requirements for Context Alterations 1009
<table>
<thead>
<tr>
<th>Instruction or Event</th>
<th>Required Before</th>
<th>Required After</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>event-based branch and rfebb</td>
<td>none</td>
<td>none</td>
<td>21</td>
</tr>
<tr>
<td>interrupt</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>hrfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Trap</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd (SF)</td>
<td>none</td>
<td>none</td>
<td>8</td>
</tr>
<tr>
<td>mtmsrd (TM)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd (TS)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (EE)</td>
<td>none</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>mtmsr[d] (PR)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtmsr[d] (FP)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (FE0,FE1)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (SE, BE)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (IR)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtmsr[d] (RI)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsr[in]</td>
<td>CSI</td>
<td>CSI</td>
<td>9</td>
</tr>
<tr>
<td>mtmspr(CIABR)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr(DEC)</td>
<td>none</td>
<td>none</td>
<td>10</td>
</tr>
<tr>
<td>mtmspr(SDR1)</td>
<td>ptesync</td>
<td>CSI 3,4,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(CTRL)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr(HDEC)</td>
<td>none</td>
<td>none</td>
<td>10</td>
</tr>
<tr>
<td>mtmspr(IAMR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr(RMOR)</td>
<td>none</td>
<td>CSI 14,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(HRMOR)</td>
<td>none</td>
<td>CSI 9,13,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(LPCR)</td>
<td>none</td>
<td>CSI 13,14</td>
<td></td>
</tr>
<tr>
<td>mtmspr(LPIDR)</td>
<td>CSI</td>
<td>CSI 7,12,16,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(PCR)</td>
<td>none</td>
<td>CSI 19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(TEXASR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr(TFHAR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr(DPDES)</td>
<td>none</td>
<td>CSI 19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(DHDES)</td>
<td>none</td>
<td>CSI 19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(BESCR)</td>
<td>none</td>
<td>CSI 18,20</td>
<td></td>
</tr>
<tr>
<td>mtmspr(FSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr(HFSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>Perf. Mon.</td>
<td>none</td>
<td>CSI 17,20</td>
<td></td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>{ptesync, CSI}</td>
<td>6,7,9</td>
</tr>
</tbody>
</table>

Table 4: Synchronization requirements for data access}

<table>
<thead>
<tr>
<th>Instruction or Event</th>
<th>Required Before</th>
<th>Required After</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>event-based branch and rfebb</td>
<td>none</td>
<td>none</td>
<td>21</td>
</tr>
<tr>
<td>interrupt</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>hrfid</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Trap</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd (SF)</td>
<td>none</td>
<td>none</td>
<td>8</td>
</tr>
<tr>
<td>mtmsrd (TM)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsrd (TS)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (EE)</td>
<td>none</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>mtmsr[d] (PR)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtmsr[d] (FP)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (FE0,FE1)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (SE, BE)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr[d] (IR)</td>
<td>none</td>
<td>none</td>
<td>9</td>
</tr>
<tr>
<td>mtmsr[d] (RI)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsr[in]</td>
<td>CSI</td>
<td>CSI</td>
<td>9</td>
</tr>
<tr>
<td>mtmspr(CIABR)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr(DEC)</td>
<td>none</td>
<td>none</td>
<td>10</td>
</tr>
<tr>
<td>mtmspr(SDR1)</td>
<td>ptesync</td>
<td>CSI 3,4,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(CTRL)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr(HDEC)</td>
<td>none</td>
<td>none</td>
<td>10</td>
</tr>
<tr>
<td>mtmspr(IAMR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr(RMOR)</td>
<td>none</td>
<td>CSI 14,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(HRMOR)</td>
<td>none</td>
<td>CSI 9,13,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(LPCR)</td>
<td>none</td>
<td>CSI 13,14</td>
<td></td>
</tr>
<tr>
<td>mtmspr(LPIDR)</td>
<td>CSI</td>
<td>CSI 7,12,16,19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(PCR)</td>
<td>none</td>
<td>CSI 19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(TEXASR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr(TFHAR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmspr(DPDES)</td>
<td>none</td>
<td>CSI 19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(DHDES)</td>
<td>none</td>
<td>CSI 19</td>
<td></td>
</tr>
<tr>
<td>mtmspr(BESCR)</td>
<td>none</td>
<td>CSI 18,20</td>
<td></td>
</tr>
<tr>
<td>mtmspr(FSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmspr(HFSCR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>Perf. Mon.</td>
<td>none</td>
<td>CSI 17,20</td>
<td></td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>{ptesync, CSI}</td>
<td>6,7,9</td>
</tr>
</tbody>
</table>

Table 5: Synchronization requirements for instruction fetch and/or execution
Notes:

1. The effect of changing the EE bit is immediate, even if the `mtmsr[d]` instruction is not context synchronizing (i.e., even if L=1).
   - If an `mtmsr[d]` instruction sets the EE bit to 0, neither an External interrupt a Decrementer interrupt nor a Performance Monitor interrupt occurs after the `mtmsr[d]` is executed.
   - If an `mtmsr[d]` instruction changes the EE bit from 0 to 1 when an External, Decrementer, Performance Monitor or higher priority exception exists, the corresponding interrupt occurs immediately after the `mtmsr[d]` is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the `mtmsr[d]` instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after `mtmsr[d]` is executed as long as the thread remains in hypervisor state.
   - If the hypervisor executes an `mtmsr[d]` instruction that changes the EE bit from 0 to 1 when a Hypervisor Decrementer or higher priority exception exists, the corresponding interrupt occurs immediately after the `mtmsr[d]` instruction is executed, and before the next instruction is executed, provided HDICE is 1.

2. Synchronization requirements for this instruction are implementation-dependent.

3. SDR1 must not be altered when MSR_DR=1 or MSR_IR=1; if it is, the results are undefined.

4. A `ptesync` instruction is required before the `mtspr` instruction because (a) SDR1 identifies the Page Table and thereby the location of Reference and Change bits, and (b) on some implementations, use of SDR1 to update Reference and Change bits may be independent of translating the virtual address. (For example, an implementation might identify the PTE in which to update the Reference and Change bits in terms of its offset in the Page Table, instead of its real address, and then add the Page Table address from SDR1 to the offset to determine the real address at which to update the bits.) To ensure that Reference and Change bits are updated in the correct Page Table, SDR1 must not be altered until all Reference and Change bit updates associated with address translations that were performed, by the thread executing the `mtspr` instruction, before the `mtspr` instruction is executed have been performed with respect to that thread. A `ptesync` instruction guarantees this synchronization of Reference and Change bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

5. For data accesses, the context synchronizing instruction before the `tlbi, tlbiel, or tlbia` instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the `tlbi, tlbiel, or tlbia` instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated.

   (If it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a `ptesync` instruction must also be used, either before or after the `tlbi, tlbiel, or tlbia` instruction. These effects of the `ptesync` instruction are described in the last paragraph of Note 8.)

6. The notation "([ptesync,CSI])" denotes an instruction sequence. Other instructions may be inter-leaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the `Store` instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the `Store` instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the `Store` instruction will be performed using the old contents of the PTE.

   The `ptesync` instruction after the `Store` instruction ensures that all searches of the Page Table that are performed after the `ptesync` instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the `ptesync` instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

   The `ptesync` instruction also ensures that all storage accesses associated with instructions preceding the `ptesync` instruction, and all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the `ptesync` instruction, before the `ptesync` instruction is executed, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the `ptesync` instruction.
**sync** instruction are performed with respect to that thread or mechanism.

7. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect.

Section 5.10 gives examples of using **tlbie**, **Store**, and related instructions to maintain the Page Table, in both multi-threaded environments and environments consisting of only a single-threaded processor.

---

### Programming Note

In a multi-threaded system, if software locking is used to help ensure that the requirements described in Section 5.10 are satisfied, the **lwsync** instruction near the end of the lock acquisition sequence (see Section B.2.1.1 of Book II) may naturally provide the context synchronization that is required before the alteration.

8. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR_SF from 1 to 0, the **mtmsrd** instruction must have an effective address that is less than $2^{32} - 4$. Furthermore, when changing MSR_SF from 0 to 1, the **mtmsrd** instruction must not be at effective address $2^{32} - 4$ (see Section 5.3.2 on page 889).

9. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

---

### Programming Note

If it is desired to set MSR_IR to 1 early in an operating system interrupt handler, advantage can sometimes be taken of the fact that EA03 are ignored when forming the real address when address translation is disabled and MSR_RV = 0. For example, if address translation resources are set such that effective address 0x000_0000_0000_0000 maps to real address 0x000_0000_0000_0000 when address translation is enabled, where n is an arbitrary 4-bit value, the following code sequence, in real page 0, can be used early in the interrupt handler.

```
la rx, target
li ry, 0x0n000
sldiry, ry, 48
or rx, rx, ry # set high-order nibble of target
               addr to 0xn
mtctrrx
bcctr  # branch to target
```

targ: mfsmsrrx
orirx, rx, 0x0020
mtmsrdrx # set MSR[IR] to 1

The **mtmsrd** does not cause an implicit branch in real address space because the real address of the next sequential instruction is independent of MSR_IR. Using **mtmsrd**, rather than **rtid** (or similar context synchronizing instruction that alters the control flow), may yield better performance on some implementations.

(Variations on the technique are possible. For example, the target instruction of the **bcctr** can be in arbitrary real page P, where P is a 48-bit value, provided that effective address 0xn || P || 0x000 maps to real address P || 0x000 when address translation is enabled.)

10. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.

11. If an **slbamte** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the **slbamte** must be preceded by an **slbie** (or **slbia**) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the **slbie** and the **slbamte**, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the **slbamte**.

No **slbie** (or **slbia**) is needed if the **slbamte** instruction replaces a valid SLB entry with a mapping of a
11. different ESID (e.g., to satisfy an SLB miss). However, the \textit{slbia} is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

12. The context synchronizing instruction before the \textit{mtspr} instruction ensures that the LPIDR is not altered out-of-order. (Out-of-order alteration of the LPIDR could permit the requirements described in Section 5.10.1 to be violated. For the same reason, such a context synchronizing instruction may be needed even if the new LPID value is equal to the old LPID value.)

See also Chapter 2. “Logical Partitioning (LPAR) and Thread Control” on page 843 regarding moving a thread from one partition to another.

13. When the RMOR or HRMOR is modified, or the VC, VRMASD, or RMLS fields of the LPCR are modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of these registers or fields (i.e., the contents immediately before the modification). The \textit{slbia} instruction can be used to invalidate all such implementation-specific lookaside information.

14. A context synchronizing instruction or event that is executed or occurs when LPCR\textsubscript{MER} = 1 does not necessarily ensure that the exception effects of LPCR\textsubscript{MER} are consistent with the contents of LPCR\textsubscript{MER}. See Section 2.2.

15. This line applies regardless of which SPR number (13 or 29) is used for the AMR.

16. LPIDR must not be altered when MSR\textsubscript{DR}=1 or MSR\textsubscript{IR}=1; if it is, the results are undefined.

17. This line applies to the following Performance Monitor SPRs: PMC1-6, MMCR0, MMCR1, MMCR2, and MMCRA.

18. This line applies to all SPR numbers that access the BESCR (800-803, 806).

19. There are additional software synchronization requirements when an \textit{mtspr} instruction modifies this SPR in a multi-threaded environment. See Section 2.8.

20. As an alternative to a CSI, the execution of an \textit{rfebb} instruction or the occurrence of an event-based branch is sufficient to provide the necessary synchronization.

21. These instructions and events, with the exception of nested \texttt{tbegin} and \texttt{tend}, TM instructions that except or are described to be treated as noops, \textit{Conditional Abort} instructions that do not abort, and events and \textit{rfebb} instructions for which the event did not take place in Transactional state, will change MSR\textsubscript{TS}. No explicit synchronization is required.
Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book III. Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the `mtspr` and `mfspr` instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, and for the Move From Time Base instruction defined in Book II.

The `mtspr` and `mfspr` instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the Move From Time Base instruction, which specifies the portion of the Time Base as a numeric operand.

**Note:** `mftb` serves as both a basic and an extended mnemonic. The Assembler will recognize an `mftb` mnemonic with two operands as the basic form, and an `mftb` mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

---

**Programming Note**

The extended mnemonics in Table 6 for SPRs associated with the Performance Monitor facility are based on the definitions in Chapter 9.

Other versions of Performance Monitor facilities used different sets of SPR numbers (all 32-bit PowerPC implementations used a different set, and some early Power ISA implementations used yet a different set).
<table>
<thead>
<tr>
<th>Special Purpose Register</th>
<th>Move To SPR</th>
<th>Move From SPR(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point Exception Register</td>
<td>mtxe Rx, mtspr 1,Rx</td>
<td>mfxer Rx, mfspr Rx,1</td>
</tr>
<tr>
<td>Link Register</td>
<td>mtlr Rx, mtspr 8,Rx</td>
<td>mflr Rx, mfspr Rx,8</td>
</tr>
<tr>
<td>Count Register</td>
<td>mtcn Rx, mtspr 9,Rx</td>
<td>mfcn Rx, mfspr Rx,9</td>
</tr>
<tr>
<td>Data Stream Control Register</td>
<td>mtdscr Rx, mtspr 17,Rx</td>
<td>mfdscr Rx, mfspr Rx,17</td>
</tr>
<tr>
<td>Data Storage Interrupt Status Register</td>
<td>mtdsisr Rx, mtspr 18,Rx</td>
<td>mfsdr Rx, mfspr Rx,18</td>
</tr>
<tr>
<td>Data Address Register</td>
<td>mtdar Rx, mtspr 19,Rx</td>
<td>mfdar Rx, mfspr Rx,19</td>
</tr>
<tr>
<td>Decrementer</td>
<td>mtdec Rx, mtspr 22,Rx</td>
<td>mfdrc Rx, mfspr Rx,22</td>
</tr>
<tr>
<td>Storage Description Register 1</td>
<td>mtsdr1 Rx, mtspr 25,Rx</td>
<td>mfsdr1 Rx, mfspr Rx,25</td>
</tr>
<tr>
<td>Save/Restore Register 0</td>
<td>mtsr0 Rx, mtspr 26,Rx</td>
<td>mfsr0 Rx, mfspr Rx,26</td>
</tr>
<tr>
<td>Save/Restore Register 1</td>
<td>mtsr1 Rx, mtspr 27,Rx</td>
<td>mfsr1 Rx, mfspr Rx,27</td>
</tr>
<tr>
<td>Come-From Address Register</td>
<td>mtcfar Rx, mtspr 28,Rx</td>
<td>mfcfar Rx, mfspr Rx,28</td>
</tr>
<tr>
<td>IC</td>
<td>mtic Rx, mtspr 48,Rx</td>
<td>mftc Rx, mfspr Rx,48</td>
</tr>
<tr>
<td>CTRL</td>
<td>mtctrl Rx, mtspr 152,Rx</td>
<td>mftc Rx, mfspr Rx,152</td>
</tr>
<tr>
<td>UAMOR</td>
<td>mtuamor Rx, mtspr 157,Rx</td>
<td>mftc Rx, mfspr Rx,157</td>
</tr>
<tr>
<td>Special Purpose Registers G0 through G3</td>
<td>mtsprg n,Rx, mtspr 272+n,Rx</td>
<td>mfsprg Rx,n, mfspr Rx,272+n</td>
</tr>
<tr>
<td>Time Base [Lower]</td>
<td>mttbl Rx, mtspr 284,Rx</td>
<td>mftb Rx, mfspr Rx,284</td>
</tr>
<tr>
<td>Time Base Upper</td>
<td>mttbu Rx, mtspr 285,Rx</td>
<td>mftbuc Rx, mfspr Rx,285</td>
</tr>
<tr>
<td>Time Base Upper 40</td>
<td>mttbu40 Rx, mtspr 286,Rx</td>
<td>-</td>
</tr>
<tr>
<td>VTB</td>
<td>mtvbtb Rx, mtspr 849,Rx</td>
<td>-</td>
</tr>
<tr>
<td>Processor Version Register</td>
<td>-</td>
<td>mtpr Rx, mfspr Rx,1023</td>
</tr>
<tr>
<td>HMER</td>
<td>mthmer Rx, mtspr 336,Rx</td>
<td>mhfmer Rx, mfspr Rx,336</td>
</tr>
<tr>
<td>HMEER</td>
<td>mthmeer Rx, mtspr 337,Rx</td>
<td>mhfmeer Rx, mfspr Rx,337</td>
</tr>
<tr>
<td>AMOR</td>
<td>mtamor Rx, mtspr 349,Rx</td>
<td>mftamor Rx, mfspr Rx,349</td>
</tr>
<tr>
<td>MMCRA</td>
<td>mtmmcra Rx, mtspr 786,Rx</td>
<td>mmmcr Ra, mfspr Rx,786</td>
</tr>
<tr>
<td>PMC1</td>
<td>mtpmc1 Rx, mtspr 787,Rx</td>
<td>mfpmc1 Rx, mfspr Rx,787</td>
</tr>
<tr>
<td>PMC2</td>
<td>mtpmc2 Rx, mtspr 788,Rx</td>
<td>mfpmc2 Rx, mfspr Rx,788</td>
</tr>
<tr>
<td>PMC3</td>
<td>mtpmc3 Rx, mtspr 789,Rx</td>
<td>mfpmc3 Rx, mfspr Rx,789</td>
</tr>
<tr>
<td>PMC4</td>
<td>mtpmc4 Rx, mtspr 790,Rx</td>
<td>mfpmc4 Rx, mfspr Rx,790</td>
</tr>
<tr>
<td>PMC5</td>
<td>mtpmc5 Rx, mtspr 791,Rx</td>
<td>mfpmc5 Rx, mfspr Rx,791</td>
</tr>
<tr>
<td>PMC6</td>
<td>mtpmc6 Rx, mtspr 792,Rx</td>
<td>mfpmc6 Rx, mfspr Rx,792</td>
</tr>
<tr>
<td>MMCR0</td>
<td>mtmmcr0 Rx, mtspr 795,Rx</td>
<td>mfmcr0 Rx, mfspr Rx,795</td>
</tr>
<tr>
<td>MMCR1</td>
<td>mtmmcr1 Rx, mtspr 798,Rx</td>
<td>mfmcr1 Rx, mfspr Rx,798</td>
</tr>
<tr>
<td>PPR</td>
<td>mtppr Rx, mtspr 896,Rx</td>
<td>mftprr Rx, mfspr Rx,896</td>
</tr>
<tr>
<td>PPR32</td>
<td>mtppr32 Rx, mtspr 898,Rx</td>
<td>mftprr32 Rx, mfspr Rx,898</td>
</tr>
<tr>
<td>Processor Identification Register</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^1\) The \textit{mftb} instruction is Category: Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an \textit{mfspr} instruction for the mftb and mftbu extended mnemonics; see the corresponding Assembler Note in the \textit{mftb} instruction description (see Section 6.2.1 of Book II).
Appendix B. Interpretation of the DSISR as Set by an Alignment Interrupt

For most causes of Alignment interrupt, the interrupt handler will emulate the interrupting instruction. To do this, it needs the following characteristics of the interrupting instruction:

- Load or store
- Length (halfword, word, doubleword)
- String, multiple, or elementary
- Fixed-point or floating-point
- Update or non-update
- Byte reverse or not
- Is it dcbz?

The Power ISA optionally provides this information by setting bits in the DSISR that identify the interrupting instruction type. It is not necessary for the interrupt handler to load the interrupting instruction from storage. The mapping is unique except for a few exceptions that are discussed below. The near-uniqueness depends on the fact that many instructions, such as the fixed- and floating-point arithmetic instructions and the one-byte loads and stores, cannot cause an Alignment interrupt.

See Section 6.5.8 for a description of how the opcode and extended opcode are mapped to a DSISR value for an X-, D-, or DS-form instruction that causes an Alignment interrupt.

The table on the next page shows the inverse mapping: how the DSISR bits identify the interrupting instruction. The following notes are cited in the table.

1. For cases in which multiple instructions can give the same value for the DSISR bits (44:45 47:53) that are derived from the opcode, the Alignment interrupt handler should load the instruction from storage, using the effective address in SRR0, and treat the instruction appropriately (e.g., emulate the instruction, treat the case as a programming error, etc.). For example, if lbarx or lwarx causes an Alignment interrupt, it should not be emulated, but instead should be treated as a programming error.

2. These are distinguished by DSISR bits 44:45, which are not shown in the table.

The interrupt handler has no need to distinguish between an X-form instruction and the corresponding D- or DS-form instruction if one exists, and vice versa. Therefore two such instructions may yield the same DSISR value (all 32 bits). For example, stw and stwx may both yield either the DSISR value shown in the following table for stw, or that shown for stwx.
Power ISA™ - Book III-S
If DSISR 47:53 is:

then it is either X-form opcode:

or D/DS/DQ/DQE-form opcode:

so the instruction is:
Book III-E:

Power ISA Operating Environment Architecture - Embedded Environment [Category: Embedded]
Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 32-Bit Implementations

Though the specifications in this document assume a 64-bit implementation, 32-bit implementations are permitted as described in Appendix C, “Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations” on page 1249.

1.3 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system auxiliary processor enabled exception error handler” substitute “Auxiliary Processor Enabled Exception type Program interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt” or Data TLB Error interrupt as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Illegal Instruction exception type Program interrupt” or “Unimplemented Operation exception type Program interrupt”, as appropriate.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt” or “Instruction TLB Error”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction exception type Program interrupt”.
- For “system service program” substitute “System Call interrupt”.
- For “system trap handler” substitute “Trap type Program interrupt”.

1.3.1 Definitions and Notation

The definitions and notation given in Books I and II are augmented by the following.

- Threaded processor, single-threaded processor, thread

A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. The only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor resources.

- Thread enabled, thread disabled

A thread can be enabled or disabled. When enabled, the thread can prefetch and execute instructions; when disabled, prefetched instructions are discarded, and the thread cannot prefetch or execute instructions.

- Performed

An explicit modification, by a thread T1, of a shared SPR (using mtsp) or an entry in a shared TLB (using tlbwe), is performed with respect to thread T2 when a read of the shared SPR or TLB
entry (using `mfspr` or `tlbre` respectively) by T2 will return the result of the modification (or of a subsequent modification). For T2, the effects of such a modification having been performed with respect to T2 are the same as if the `mfspr` or `tlbre` were in T2’s instruction stream at the point at which the modification was performed with respect to T2. T1 and T2 may be any threads that share the SPR or TLB with one another, and may be the same thread.

- real page

A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size may range from 1KB to 1TB [MAV=1.0] or to 2TB [MAV=2.0].

- [MAV=x.x]

Instructions and facilities are considered part of all MMU architecture versions unless otherwise marked. If a facility or section is marked with a specific MMU Architecture version x.x, that facility or all material in that section and its subsections are considered part of the specific MMU architecture version.

- "must"

If the Embedded.Hypervisor category is not supported and privileged software violates a rule that is stated using the word “must”, the results are undefined. If the Embedded.Hypervisor category is supported, the following applies.

- If hypervisor software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage using a TLB entry with a TGS value of 0, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc.

- If supervisor software violates the requirements for storage control bit values or their alteration, the result of accessing the associated storage is undefined.

- In other cases of violation of a rule that is stated using the word “must”, the results are boundedly undefined unless otherwise stated.

---

**Programming Note**

Contrary to the general principle of partition isolation, the result of accessing storage associated with violations of the requirements for storage control bit values and their alteration is specified as “undefined”. This is the only case where a guest operating system can cause “undefined” results. Embedded architecture does this as a hardware simplification because of the limited amount of code involved with storage control bits and because operating systems used in the Embedded environment can be tested and then controlled.

- context of a program

The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR, of certain lookaside buffers, such as the TLB, and of other resources.

- exception

An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

- interrupt

The act of changing the machine state in response to an exception, as described in Chapter 7. “Interrupts and Exceptions” on page 1145.

- trap interrupt

An interrupt that results from execution of a `Trap` instruction.

- Additional exceptions to the sequential execution model, beyond those described in the section entitled “Instruction Fetching” in Book I, are the following.
  - A reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 12. “Synchronization Requirements for Context Alterations” on page 1235). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.
Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.

- hypervisor privileged (or hypervisor-privileged)

If category E.HV is implemented, this term describes an instruction, register, or facility that is available only when the thread is in hypervisor state. Otherwise, this term describes an instruction, register, or facility that is available only when the thread is in supervisor state.

- privileged state and supervisor state

Used interchangeably to refer to a state in which privileged facilities are available.

- guest state

A state used to run software under control of a hypervisor program in which hypervisor-privileged facilities are not available.

- guest supervisor state

A state which is in both the guest state and the supervisor state.

- problem state and user mode

Used interchangeably to refer to a state in which privileged facilities are not available.

- volatile

Bits in a register or array (e.g., TLB) are considered volatile if they may change even if not explicitly modified by software.

- directed

In a hypervisor system, the attribute of an interrupt that execution occurs in the guest supervisor or hypervisor state as described in Section 2.3.1, “Directed Interrupts”.

- \(/, //, ///, ...\) denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

- \(??, ??, ???, ...\) denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

### 1.3.2 Reserved Fields

Some fields of certain architected registers may be written to automatically by the hardware, e.g., Reserved bits in System Registers. When the hardware writes to such a register, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.

- Contents of reserved fields are either preserved or written as zero.

The reader should be aware that reading and writing of some of these registers (e.g., the MSR) can occur as a side effect of processing an interrupt and of returning from an interrupt, as well as when requested explicitly by the appropriate instruction (e.g., mtmsr instruction).

### 1.4 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions fall into the following classes:

- instructions executed in the Branch Facility
- instructions executed in the Fixed-Point Facility
- instructions executed in the Floating-Point Facility
- instructions executed in the Vector Facility
- instructions executed in an Auxiliary Processor
- other instructions

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book I may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions executed in an Auxiliary Processor are implementation-dependent. Instructions related to the supervisor mode, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

### 1.5 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR<sub>FP</sub>=0 (Floating-Point Unavailable interrupt)
- execution of an instruction that causes a debug event (Debug interrupt).
- the execution of an auxiliary processor instruction when the auxiliary processor is unavailable (Auxiliary Processor Unavailable interrupt)
- the execution of a Vector, SPE, or Embedded Floating-Point instruction when MSR<sub>SPV</sub>=0 (SPE/Embedded Floating-Point/Vector Unavailable interrupt)
1.6 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.6.1 Context Synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below. Such instructions and events are collectively called context synchronizing operations. The context synchronizing operations include the dh instruction, the isync instruction, the System Linkage instructions, and most interrupts (see Section 7.1). Also, the combination of disabling and enabling a thread is context-synchronizing for the thread being enabled (See Section 3).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.

2. The operation is not initiated or, in the case of dh, isync does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.

3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated.

4. If the operation directly causes an interrupt (e.g., sc directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 7.9, “Exception Priorities” on page 1190).

5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 6.5, “Performing Operations Out-of-Order”.)

The operation ensures that all explicit modifications of shared SPRs (mtspr) and shared TLBs (tlbwe), caused by instructions that precede the operation, have been performed with respect to all other threads that share the SPR or TLB.

---

**Programming Note**

The context established by a context synchronizing instruction includes modifications to certain resources that were performed with respect to the context synchronizing thread before the operation was initiated. The resources in this case include shared SPRs that contain program context such as LPIDR as well as TLBs shared by other threads.

---

**Programming Note**

A context synchronizing operation is necessarily execution synchronizing; see Section 1.6.2.

Unlike the Synchronize instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for isync (and sync; see Section 1.6.2) because all other execution synchronizing operations also alter context.

1.6.2 Execution Synchronization

An instruction is execution synchronizing if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.6.1). sync is treated like isync with respect to item 2. The execution synchronizing instructions are sync, mtmsr and all context synchronizing instructions.

---

**Programming Note**

Unlike a context synchronizing operation, an execution synchronizing instruction does not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.
Chapter 2. Logical Partitioning [Category: Embedded.Hypervisor]

2.1 Overview

The Embedded.Hypervisor category permits threads and portions of real storage to be assigned to logical collections called partitions, such that a program executing in one partition cannot interfere with any program executing in a different partition. This isolation can be provided for both problem state and privileged state programs, by using a layer of trusted software, called a hypervisor program (or simply a “hypervisor”), and the resources provided by this facility to manage system resources. The collection of software that runs in a given partition and its associated resources is called a guest. The guest normally includes an operating system (or other system software) running in privileged state and its associated processes running in the problem state under the management of the hypervisor. The thread is in the guest state when a guest is executing and is in the hypervisor state when the hypervisor is executing. The thread is executing in the guest state when MSRGS=1.

The number of partitions supported is implementation-dependent.

A thread is assigned to one partition at any given time. A thread can be assigned to any given partition without consideration of the physical configuration of the system (e.g. shared registers, caches, organization of the storage hierarchy), except that threads that share certain hypervisor resources may need to be assigned to the same partition. Additionally, certain resources may be utilized by the guest at the discretion of the hypervisor. Such usage may cause interference between partitions and the hypervisor should allocate those resources accordingly. The primary registers and facilities used to control Logical Partitioning are listed below and described in the following subsections. Other facilities associated with Logical Partitioning are described within the appropriate sections within this Book.

An instruction that is hypervisor privileged must be execute in the hypervisor state (MSRGS PR = 0b00). If an attempt is made to execute a hypervisor-privileged instruction in the guest supervisor state (MSRGS PR = 0b10), an Embedded Hypervisor Privilege exception occurs. A register that is hypervisor privileged may only be accessed in the hypervisor state (MSRGS PR = 0b00). If a hypervisor-privileged register is accessed in the guest supervisor state (MSRGS PR = 0b10), an Embedded Hypervisor Privilege exception occurs.

When MSRGS PR = 0b01 or MSRGS PR = 0b11, the thread is in problem (user) state. The resources (instructions and registers) that are available are generally the same when MSRP R = 0b1 regardless of the state of MSRGS; however when MSRGS PR = 0b11 some interrupts are directed to the guest supervisor state. When MSRGS PR = 0b01 interrupts are always directed to the hypervisor (see Section 2.3.1, “Directed Interrupts”).

Category Embedded.Hypervisor changes the operating system programming model to allow for easier virtualization, while retaining a default backward compatible mode in which an operating system written for hardware not implementing this category will still operate as before without using the Logical Partitioning facilities.

Category Embedded.Hypervisor requires that Category: Embedded.Processor Control is also supported.

2.2 Registers

Registers specific to Logical Partitioning and hypervisor control are defined in this section. Other registers which are hypervisor privileged or have hypervisor-only fields appear and are described in other sections in this Book.

2.2.1 Register Mapping

To facilitate better performance for operating systems executing in the guest supervisor state, some Special Purpose Register (SPR) accesses are redirected to analogous guest-state SPRs. An SPR is said to be mapped if this redirection takes place when executing in guest supervisor state. These guest-state SPRs separate performance critical state of the hypervisor and the operating system executing in guest supervisor state.
state. The mapping of these register accesses allows the same programming model to be used for an operating system running in the guest supervisor state or in the hypervisor state.

For example, when a $\text{mtspr}\ SRR0,r5$ instruction is executed in guest supervisor state, the access to SRR0 is mapped to GSRR0. This produces the same operation as executing $\text{mtspr}\ GSRR0,r5$.

---

**Programming Note**

Since accesses to the mapped SPRs are automatically mapped to the appropriate guest-accessible SPR, guest supervisor software should use the original SPRs for accessing these registers (i.e. SRR0, not GSRR0). This facilitates using the same code in hypervisor or guest state.

SPR accesses that are mapped in guest supervisor state are listed in Table 1.

<table>
<thead>
<tr>
<th>SPR Accessed</th>
<th>SPR Mapped to</th>
<th>Type of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>GDEC</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>DECAR</td>
<td>GDECAR</td>
<td>$\text{mtspr}$</td>
</tr>
<tr>
<td>TCR</td>
<td>GTCR</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>TSR</td>
<td>GTSR</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>SRR0</td>
<td>GSRR0</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>SRR1</td>
<td>GSRR1</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>EPR</td>
<td>GEPR</td>
<td>$\text{mfspr}$</td>
</tr>
<tr>
<td>ESR</td>
<td>GESR</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>DEAR</td>
<td>GDEAR</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>PIR</td>
<td>GPIR</td>
<td>$\text{mfspr}$</td>
</tr>
<tr>
<td>SPRG0</td>
<td>GSPRG0</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>SPRG1</td>
<td>GSPRG1</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>SPRG2</td>
<td>GSPRG2</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
<tr>
<td>SPRG3</td>
<td>GSPRG3</td>
<td>$\text{mtspr},\text{mfspr}$</td>
</tr>
</tbody>
</table>

1 If an implementation permits problem state read access to SPRG3, the problem state read access is remapped to GSPRG3.

Table 1: Mapped SPRs

---

### 2.2.2 Logical Partition Identification Register (LPIDR)

The Logical Partition Identification Register (LPIDR) contains the Logical Partition ID (LPID) currently in use for the thread. The format of the LPIDR is shown in Figure 1 below.

---

**2.3 Interrupts and Exceptions**

#### 2.3.1 Directed Interrupts

Category Embedded.Hypervisor introduces new interrupt semantics. Interrupts are directed to either the guest state or the hypervisor state. The state to which interrupts are directed determines which SPRs are used to form the vector address, which save/restore registers are used to capture the thread state at the time of the interrupt, and which registers are used to post exception status.

- If IVORs [Category: Embedded.Phased-Out] are supported, interrupts directed to the guest state use the Guest Interrupt Vector Prefix Register (GIVPR) to determine the high-order 48 bits of the vector address and use Guest Interrupt Vector Registers (GIVORs) to provide the low-order 16 bits (of which the last 4 bits are 0).
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, interrupts directed to the guest state use the Guest Interrupt Vector Prefix Register (GIVPR) to determine the high-order 52 bits of the vector address and use the 12-bit exception vector offsets (described in Section 7.2.15) to provide the low-order 12 bits (of which the last 5 bits are 0).
- If IVORs [Category: Embedded.Phased-Out] are supported, interrupts directed to the Embedded hypervisor state use the IVPR for the upper 48 bits of the address and the IVORs for the lower 16 bits of the address.
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, interrupts directed to the Embedded hypervisor state use one of the following for the interrupt vector address.
  - If the Machine Check Interrupt Vector Prefix Register (see Section 7.2.18.4) is supported and the interrupt is a Machine Check, MCIVPR provides the high-order 52 bits of the vector address and the 12-bit exception vector offsets (described in Section 7.2.15) provides the low-order 12 bits (of which the last 5 bits are 0).
  - Otherwise, IVPR provides the high-order 52 bits of the vector address and the 12-bit exception vector offsets (described in Section
7.2.15) provides the low-order 12 bits (of which the last 5 bits are 0).

Interrupts that are directed to the guest state use GSRR0/GSRR1 registers to save the context at interrupt time. Interrupts directed to the embedded hypervisor state use SRR0/SRR1, with the exception of Guest Processor Doorbell interrupts which use GSRR0/GSRR1.

All interrupts are directed to the hypervisor except when the processor is already in guest state (MSRGS = 1) and:
- The interrupt is a system call and the LEVEL field is 0.
- The interrupt is a Data TLB Error, Instruction TLB Error, Data Storage Interrupt, Instruction Storage Interrupt, or External Input Interrupt and the corresponding bit in the EPCR to direct these interrupts to guest state is a 1 and the interrupt is not caused by either a Virtualization Fault or a TLB Ineligible Exception.

### 2.3.2 Hypervisor Service Interrupts

Two interrupts exist as mechanisms for the hypervisor to provide services to the guest.

The Embedded Hypervisor Privilege Interrupt occurs when guest supervisor state attempts execution of a hypervisor-privileged instruction or attempts to access a hypervisor-privileged resource. This can be used by the hypervisor to provide virtualization services for the guest. The Embedded Hypervisor Privilege Interrupt is described in Section 7.6.31, “Embedded Hypervisor Privilege Interrupt [Category: Embedded.Hypervisor]”. An Embedded Hypervisor Privilege Interrupt will also occur if a `ehpriv` instruction is executed regardless of the thread state.

The Embedded Hypervisor System Call Interrupt occurs when an `sc` instruction is executed and LEV = 1. The `sc` instruction is described in Section 4.3.1, “System Linkage Instructions”.

### 2.4 Instruction Mapping

When executing in the guest supervisor state (MSRGS PR = 0b10), execution of an `rfi` instruction is mapped to `rfgi` and the `rfgi` instruction is executed in place of the `rfi`. The mapping of these instructions allows the same programming model to be used for an operating system running in the guest supervisor state or in the hypervisor state.
Chapter 3. Thread Control [Category: Embedded Multi-Threading]

3.1 Overview

The Thread Control facility permits the hypervisor to control and monitor the execution, priority, and other aspects of threads.

3.2 Thread Identification Register (TIR)

The layout of the TIR is shown in below.

Figure 2. Thread Identification Register

The TIR is a 64-bit read-only register that can be used to distinguish the thread from other threads on a multi-threaded processor. Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which $TIR = n$ is referred to as “thread n.”

The TIR is hypervisor privileged.

3.3 Thread Enable Register (TEN)

The layout of the TEN is shown in below.

Figure 3. Thread Enable Register

The TEN is a 64-bit register. For $t < T$, where $T$ is the number of threads supported by the implementation, bit 63-t corresponds to thread t. When $TEN_{63-t}$ is 0, thread t is disabled. When $TEN_{63-t}$ is 1, thread t is enabled.

Software is permitted to write any value to bits 0:63-T; a subsequent reading of these bits always returns 0.

The TEN can be accessed using two SPR numbers.

- When SPR 438 (Thread Enable Set, or TENS) is written, threads for which the corresponding bit in TENS is 1 are enabled; threads for which the corresponding bit in TENS is 0 are unaffected.

- When SPR 439 (Thread Enable Clear, or TENC) is written, threads for which the corresponding bit in TENC is 1 are disabled; threads for which the corresponding bit in TENC is 0 are unaffected.

When each SPR is read, the current value of the TEN is returned.

The TEN is hypervisor privileged.

3.4 Thread Enable Status Register (TENSR)

The layout of the TENSR is shown in below.

Figure 4. Thread Enable Status Register

32-bit implementations are limited to 32 threads. In 64-bit mode, the high order 32 bits of TEN are not modified when written. See Section 1.5.2, "Modes [Category: Embedded]", in Book I.

Software can determine the number of threads supported by the implementation by setting each progressively higher-order bit to 1, and testing whether a subsequent read returns a 1. Because this operation enables the thread, software should ensure that an acceptable instruction sequence is located at the thread's starting effective address. (See Section 8.3, "Thread State after Reset".)
The TENSR is a 64-bit read-only register. Bit 63-t of the TENSR corresponds to thread t. The contents of the TENSR are equal to the contents of the TEN, except that when TEN63-t changes from 1 to 0, TENSR63-t does not change from 1 to 0 until thread t is disabled.

The TENSR is hypervisor privileged.

Architectural Note

32-bit implementations are limited to 32 threads.

3.5 Disabling and Enabling Threads

The combination of disabling and enabling a thread is context-synchronizing for the thread being enabled. Steps 1-3 of context synchronization (see Section 1.6.1) occur as a result of the thread being disabled, and updates to SPRs and shared TLBs caused by preceding instructions executed by the thread occur. When all updates to these shared SPRs and shared TLBs have been performed with respect to all other threads on a multi-threaded processor, the TENSR bit corresponding to the disabled thread is set to 0.

Asynchronous interrupts that occur after the thread is disabled are pended until the thread is enabled.

When a thread is enabled by setting the TEN bit corresponding to the thread to 1, the thread begins execution at the next instruction to be executed when it was disabled or at the effective address specified by the INIA [Category: Embedded Multi-threading.Thread Management] if the INIA corresponding to the thread was written while the thread was disabled.

Programming Note

The architecture provides no method to make a thread's updates to shared storage visible to other threads before it is disabled. Similarly, the architecture provides no method to make updates to shared storage made while a thread is disabled visible to a thread when it is subsequently enabled.

Programming Note

When thread T1 disables other threads, Tn, it sets the TEN bits corresponding to Tn to 0s. In order to ensure that all updates to shared SPRs and shared TLBs caused by instructions being performed by threads Tn have been performed with respect to all threads on a multi-threaded processor, thread T1 reads the TENSR until all the bits corresponding to the disabled threads, Tn, are 0s.

3.6 Sharing of Multi-Threaded Processor Resources

The PVR and TEN must be shared among all threads of a multi-threaded processor. Various other resources are allowed to be shared among threads. Programs that modify shared resources must be aware of such sharing, and must allow for the fact that changes to these resources may affect more than one thread.

Resources that may be shared are grouped into the following five groups of related resources. If any of the resources in a group are shared among threads, all of the resources in the group must be shared.

- ATB, ATBL, ATBU [Category: ATB]
- IVORs [Category: Phased-Out]
- IVPR
- TB, TBL, TBU
- MMUCFG, MMUCSR0, TLB, TLBnCFG, TLBnCFG2, TLBnEPT, [Category: Embedded.Hypervisor.LRAT]: LRAT, LRATCFG, LRATCFG2

If the implementation requires all threads to be in the same partition, the following additional groups of resources may be shared. If any of the resources in a group are shared among threads, all of the resources in the group must be shared.

- DAC1, DAC2, IAC1, IAC2, IAC3
- EHCSR [Category: Embedded.Hypervisor]
- GIVORs [Category: Phased-Out]
- GIVPR [Category: Embedded.Hypervisor]
- LPIDR [Category: Embedded.Hypervisor]

Certain implementation-dependent registers, instruction and Data Caches, and implementation-dependent look-aside information may also be shared.

The set of resources that is shared is implementation-dependent.
When software executing in thread T1 writes a new value in an SPR (mtspr) that is shared with other threads, or explicitly writes to an entry in a shared TLB (tlbwe), either of the following sequences of operations can be performed in order to ensure that the write operation has been performed with respect to other threads.

Sequence 1

- Disable all other threads (see Section 3.5)
- Write to the shared SPR (mtspr) or to the shared TLB (tlbwe)
- Perform a context synchronizing operation
- Enable the previously-disabled threads

In the above sequence, the context synchronizing operation ensures that the write operation has been performed with respect to all other threads that share the SPR or TLB; the enabling of other threads ensures that subsequent instructions of the enabled threads use the new SPR or TLB value since enabling a thread is a context synchronizing operation.

Sequence 2

- All threads are put in hypervisor state and begin polling a storage flag
- The thread updating the SPR or TLB does the following:
  - Writes to the SPR (mtspr) or the TLB (tlbwe)
  - Sets a storage flag indicating the write operation was done
  - Performs a context synchronizing operation
- When other threads see the updated storage flag, they perform context synchronizing operations.

In the above sequence, the context synchronizing operation by the thread that writes to the SPR or TLB ensures that the write operation has been performed with respect to all other threads that share the SPR or TLB; the context synchronizing operation by the other threads ensure that subsequent instructions for these threads use the updated value.

### Programming Note

When software executing in thread T1 writes a new value in an SPR (mtspr) that is shared with other threads, or explicitly writes to an entry in a shared TLB (tlbwe), either of the following sequences of operations can be performed in order to ensure that the write operation has been performed with respect to other threads.

Sequence 1

- Disable all other threads (see Section 3.5)
- Write to the shared SPR (mtspr) or to the shared TLB (tlbwe)
- Perform a context synchronizing operation
- Enable the previously-disabled threads

In the above sequence, the context synchronizing operation ensures that the write operation has been performed with respect to all other threads that share the SPR or TLB; the enabling of other threads ensures that subsequent instructions of the enabled threads use the new SPR or TLB value since enabling a thread is a context synchronizing operation.

Sequence 2

- All threads are put in hypervisor state and begin polling a storage flag
- The thread updating the SPR or TLB does the following:
  - Writes to the SPR (mtspr) or the TLB (tlbwe)
  - Sets a storage flag indicating the write operation was done
  - Performs a context synchronizing operation
- When other threads see the updated storage flag, they perform context synchronizing operations.

In the above sequence, the context synchronizing operation by the thread that writes to the SPR or TLB ensures that the write operation has been performed with respect to all other threads that share the SPR or TLB; the context synchronizing operation by the other threads ensure that subsequent instructions for these threads use the updated value.

### Figure 5. Initialize Next Instruction Address Register

<table>
<thead>
<tr>
<th>Instruction Address</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62 63</td>
</tr>
</tbody>
</table>

### 3.7 Thread Management Facility

[Category: Embedded Multi-threading.Thread Management]

The thread management facility enables software to control features related to threads. The capabilities provided allow software, for a disabled thread, to specify the address of the instruction to be executed when the thread is enabled. Other implementation-dependent capabilities may also be provided.

#### 3.7.1 Initialize Next Instruction Address Registers

The Initialize Next Instruction Address (INIAn, where n = 0..63) registers are 64-bit write-only registers that can be used to specify the effective address of the instruction to be executed when a currently-disabled thread is enabled. INIAn corresponds to thread n.

The initial value of all INIAs is $0xFFFF_FFFF_FFFF_FFFC$.
3.7.2 Thread Management Instructions

Move To Thread Management Register

\textit{XFX-form}

\texttt{mttmr} \text{ TMR,RS}

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 31 & 6 & 494 & / 11 & 21 & 31 \\
\hline
\end{tabular}
\end{center}

\( n \leftarrow \text{tmr}_{5:9} || \text{tmr}_{0:4} \)

\( \text{TMR}(n) \leftarrow (\text{RS}) \)

The TMR field denotes a Thread Management Register, encoded as shown in the table below. The contents of register RS are placed into the designated Thread Management Register.

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textbf{decimal} & \textbf{TMR} & \textbf{Register Name} \\
\hline
320 & 01010 00000 & \text{INIA}_0 \\
\hline
\ldots & \ldots & \ldots \\
383 & 10111 11111 & \text{INIA}_{63} \\
\hline
\end{tabular}

\textsuperscript{1} Note that the order of the two 5-bit halves of the SPR number is reversed.
\end{center}

Figure 6. Thread Management Register Numbers

All values of the TMR field not shown in Figure 6 are implementation-specific.

An implementation only provides INIA registers corresponding to its implemented threads. Execution of this instruction specifying a TMR number that is not defined for the implementation causes an Illegal Instruction type Program interrupt if MSR\_GS\_PR=0b00.

This instruction is hypervisor privileged.

\textbf{Special Registers Altered:} \\
See above
4.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

4.2 Branch Facility Registers

4.2.1 Machine State Register

The MSR (MSR) is a 32-bit register. MSR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). This register defines the state of the thread. The MSR can also be modified by the \textit{mtmsr}, \textit{rfi}, \textit{rfci}, \textit{rfdi} [Category: Embedded.Enhanced Debug], \textit{rfmci}, \textit{rfgi} [Category: Embedded.Hypervisor], \textit{wrtee} and \textit{wrteei} instructions and interrupts. It can be read by the \textit{mfmsr} instruction.

![Machine State Register](image)

Below are shown the bit definitions for the Machine State Register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>\textit{Computation Mode} (CM)</td>
</tr>
<tr>
<td></td>
<td>0 The thread runs in 32-bit mode.</td>
</tr>
<tr>
<td></td>
<td>1 The thread runs in 64-bit mode.</td>
</tr>
<tr>
<td>33</td>
<td>Reserved</td>
</tr>
<tr>
<td>34</td>
<td>Implementation-dependent</td>
</tr>
<tr>
<td>35</td>
<td>\textit{Guest State} (GS)</td>
</tr>
<tr>
<td></td>
<td>[Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>0 The thread is in hypervisor state if MSR$_{PR}$ = 0.</td>
</tr>
<tr>
<td></td>
<td>1 The thread is in guest state.</td>
</tr>
<tr>
<td></td>
<td>MSR$_{GS}$ cannot be changed unless thread is in the hypervisor state.</td>
</tr>
</tbody>
</table>

Virtualized Implementation Note

In a virtualized implementation, MSR$_{GS}$ will be 1.

Implementation-dependent

\textbf{User Cache Locking Enable} (UCLE) [Category: Embedded Cache Locking,User Mode]

0 \textit{Cache Locking} instructions are privileged.
1 \textit{Cache Locking} instructions can be executed in user mode (MSR$_{PR}$=1).

If Category: Embedded Cache Locking,User Mode is not supported, this bit is treated as reserved.

\textbf{SP/Embedded Floating-Point/Vector Available} (SPV) [Category: Signal Processing]:

0 The thread cannot execute any SP instructions except for the \textit{brinc} instruction.
1 The thread can execute all SP instructions.

[Category: Vector]:

0 The thread cannot execute any Vector instruction.
1 The thread can execute Vector instructions.

Reserved
40  **VSX Available** (VSX)

0  The thread cannot execute any VSX instructions, including VSX loads, stores, and moves.
1  The thread can execute VSX instructions.

---

**Programming Note**

An application binary interface defined to support Category: Vector-Scalar operations should also specify a requirement that MSR. FP and MSR. VEC be set to 1 whenever MSR. VSX is set to 1.

41:45  Reserved

46  **Critical Enable** (CE)

0  Critical Input, Watchdog Timer, Guest Processor Doorbell Critical <E.HV>, and Processor Doorbell Critical interrupts are disabled.
1  Critical Input, Watchdog Timer, Guest Processor Doorbell Critical <E.HV>, and Processor Doorbell Critical interrupts are enabled.

---

[Category: Embedded.Hypervisor]

Critical level interrupts with the exception of Guest Processor Doorbell Critical are enabled regardless of the state of MSR CE when MSR GS = 1.

47  Reserved

48  **External Enable** (EE)

0  External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell <E.HV> and Embedded Performance Monitor <E.PM> interrupts are disabled.
1  External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell <E.HV>, and Embedded Performance Monitor <E.PM> interrupts are enabled.

---

[Category: Embedded.Hypervisor]

When an interrupt that is maskable by MSR EE is directed to the hypervisor state, the interrupt is enabled if MSR EE=1 or MSR GS=1 except for Guest Processor Doorbell which is enabled if MSR EE=1 and MSR GS=1. When an interrupt that is maskable by MSR EE is directed to the guest supervisor state, the interrupt is enabled if MSR EE=1 and MSR GS=1. Also, see the EXTGS bit in Section 4.2.3.

49  **Problem State** (PR)

0  The thread is in privileged state (supervisor state).
1  The thread is in problem state (user mode).

MSR PR also affects storage access control, as described in Section 6.7.6

50  **Floating-Point Available** (FP)

[Category: Floating-Point]

0  The thread cannot execute any floating-point instructions, including floating-point loads, stores and moves.
1  The thread can execute floating-point instructions.

---

51  **Machine Check Enable** (ME)

0  Machine Check interrupts are disabled.
1  Machine Check interrupts are enabled.

---

[Category: Embedded.Hypervisor]

Machine Check interrupts with the exception of Guest Processor Doorbell Machine Check are enabled regardless of the state of MSR ME when MSR GS = 1.

52  **Floating-Point Exception Mode 0** (FE0)

[Category: Floating-Point]

(See below)

53  Implementation-dependent

54  **Debug Interrupt Enable** (DE)

0  Debug interrupts are disabled.
1  Debug interrupts are enabled if DBCR0IDM=1

---

**Virtualized Implementation Note**

In a virtualized implementation, when MSR DE=1, the registers SPRG9, DSRR0, and DSRR1 are volatile.

55  **Floating-Point Exception Mode 1** (FE1)

[Category: Floating-Point]

(See below)

56  Reserved

57  Reserved

58  **Instruction Address Space** (IS)

0  The thread directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry).
1  The thread directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).

---

59  **Data Address Space** (DS)
0 The thread directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry).
1 The thread directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).

60 Implementation-dependent
61 Performance Monitor Mark (PMM) [Category: Embedded.Performance Monitor]
0 Disable statistics gathering on marked processes.
1 Enable statistics gathering on marked processes

See Appendix D for additional information.

62 Reserved
63 Reserved

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

<table>
<thead>
<tr>
<th>FE0</th>
<th>FE1</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Ignore Exceptions</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Imprecise Nonrecoverable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Imprecise Recoverable</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Precise</td>
</tr>
</tbody>
</table>

See Section 8.3 for the initial state of the MSR.

[Category: Embedded.Hypervisor]
Some bits in the MSR can only be changed when the thread is in hypervisor state or the MSRP register has been configured to allow changes in the guest supervisor state. See Section 4.2.2, "Machine State Register Protect Register (MSRP)".

**Programming Note**
A Machine State Register bit that is reserved may be altered by rfi/rdi [Category: Embedded.Enhanced Debug]/rfi [Category: Embedded.Hypervisor].

4.2.2 Machine State Register Protect Register (MSRP)

The Machine State Register Protect Register (MSRP) controls whether certain bits in the Machine State Register (MSR) can be modified in guest supervisor state. In addition, the MSRP impacts the behavior of cache locking and performance monitor instructions in guest state, as described below. The format of the MSRP is shown in Figure 8 below.

![Figure 8. Machine State Register Protect Register](image)

The MSRP is used to prevent guest supervisor state program from modifying the UCLE, DE, or PMM bits in the MSR. The MSRP bits UCLEP, DEP, and PMMP control whether the guest can change the corresponding MSR bits UCLE, DE, and PMM, respectively. When the MSRP bit associated with a corresponding MSR bit is 0, any operation in guest privileged state is allowed to modify that MSR bit, whether from an instruction that modifies the MSR, or from an interrupt which is taken in the guest supervisor state. When the MSRP bit associated with a corresponding MSR bit is 1 no operation in guest privileged state is allowed to modify that MSR bit (i.e., it remains unchanged), whether from an instruction that modifies the MSR, or from an interrupt from the guest state which is taken in the guest supervisor state.

These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:36</td>
<td>Reserved</td>
</tr>
<tr>
<td>37</td>
<td>User Cache Lock Enable Protect (UCLEP) [Category: ECL]</td>
</tr>
<tr>
<td>0</td>
<td>MSR_{UCLE} can be modified in guest supervisor state.</td>
</tr>
<tr>
<td>1</td>
<td>MSR_{UCLE} cannot be modified in guest supervisor state and guest state cache locking using dcbtls, dcbtstls, dcblc, dcblq, icbtls, icblq, and icblc is affected as described later in this section.</td>
</tr>
<tr>
<td>38:53</td>
<td>Reserved</td>
</tr>
<tr>
<td>54</td>
<td>Debug Enable Protect (DEP)</td>
</tr>
<tr>
<td>0</td>
<td>MSR_{DE} can be modified in guest supervisor state.</td>
</tr>
<tr>
<td>1</td>
<td>MSR_{DE} cannot be modified in guest supervisor state.</td>
</tr>
<tr>
<td>55:60</td>
<td>Reserved</td>
</tr>
<tr>
<td>61</td>
<td>Performance Monitor Mark Protect (PMMP) [Category: E.PM]</td>
</tr>
<tr>
<td>0</td>
<td>MSR_{PMM} can be modified in guest supervisor state.</td>
</tr>
<tr>
<td>1</td>
<td>MSR_{PMM} cannot be modified in guest supervisor state and guest state accesses to Performance Monitor Registers using mfpmr and mtpmr are affected as described later in this section.</td>
</tr>
<tr>
<td>62:63</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The MSRP is hypervisor privileged.
A context synchronizing operation must be performed following a change to MSRP to ensure that its changes are visible in the current context.

The behavior of cache locking instructions (dcbtls, dcbltsl, dcblc, dcblq, icbtl, icblq, icblc) in guest privileged state is dependent on the setting of MSRP_UCLEP. When MSRP_UCLEP = 0, cache locking instructions are permitted to execute normally in the guest privileged state. When MSRP_UCLEP = 1, cache locking instructions are not permitted to execute in the guest privileged state and cause an Embedded Hypervisor Privilege exception. [Category: ECL]

The behavior of Performance Monitor instructions (mtprm, mfpmr) is dependent on the setting of MSRP_PMMMP. When MSRP_PMMMP = 0, Performance Monitor instructions are permitted to execute normally in the guest state. When MSRP_PMMMP = 1, Performance Monitor instructions are not permitted to execute normally in the guest state. Execution of a mfpmr instruction which specifies a user Performance Monitor register produces a value of 0 in the destination GPR. In the guest supervisor state (MSRP = 0 and MSR GS = 1), execution of any mfpmr instruction or execution of a mfpmr instruction which specifies a privileged Performance Monitor Register produces an Embedded Hypervisor Privilege exception. [Category: E.PM]

### Programming Note

Setting the MSRP to 0 at initialization allows guest state access to MSR UCLE, DE, PMMP and the associated cache locking and performance monitor facilities.

#### 4.2.3 Embedded Processor Control Register (EPCR)

The Embedded Processor Control Register (EPCR) provides general controls for both privileged and hypervisor privileged facilities. The format of the EPCR is shown in Figure 1 below.

**Figure 9. Embedded Processor Control Register**

These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>External Input Interrupt Directed to Guest State (EXTGS) [Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Controls whether an External Input Interrupt is taken in the guest supervisor state or the hypervisor state.</td>
</tr>
<tr>
<td>33</td>
<td>Data TLB Error Interrupt Directed to Guest State (DTLBGS) [Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Controls whether a Data TLB Error Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state.</td>
</tr>
<tr>
<td>34</td>
<td>Instruction TLB Error Interrupt Directed to Guest State (ITLBGS) [Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Controls whether an Instruction TLB Error Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state.</td>
</tr>
<tr>
<td>35</td>
<td>Data Storage Interrupt Directed to Guest State (DSIGS) [Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Controls whether a Data Storage Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state, except for an interrupt caused by a TLB Ineligible exception &lt;E.PT&gt;.</td>
</tr>
<tr>
<td>36</td>
<td>Instruction Storage Interrupt Directed to Guest State (ISIGS) [Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Controls whether an Instruction Storage Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state.</td>
</tr>
</tbody>
</table>
state, except for an interrupt caused by a TLB ineligible exception <E.PT>.

0 Instruction Storage Interrupts that occur in the guest supervisor state are directed to the hypervisor state.
1 Instruction Storage Interrupts that occur in the guest state are directed to the guest supervisor state.

37 **Disable Embedded Hypervisor Debug** (DUVD)
   [Category: Embedded.Hypervisor]
   Controls whether Debug Events occur in the hypervisor state.
   0 Debug events can occur in the hypervisor state.
   1 Debug events, except for the Unconditional Debug Event, are suppressed in the hypervisor state. It is implementation-dependent whether the Unconditional Debug Event is suppressed.

38 **Interrupt Computation Mode** (ICM)
   [Category: 64-bit]
   If category E.HV is implemented, this bit controls the computational mode of the thread when an interrupt occurs that is directed to the hypervisor state. At interrupt time, EPCR_{ICM} is copied into MSR_{CM} if the interrupt is directed to the hypervisor state.

   If category E.HV is not implemented, then this bit controls the computational mode of the thread when any interrupt occurs. At interrupt time, EPCR_{ICM} is copied into MSR_{CM}.
   0 Interrupts will execute in 32-bit mode.
   1 Interrupts will execute in 64-bit mode.

39 **Guest Interrupt Computation Mode** (GICM)
   [Category: Embedded.Hypervisor]
   [Corequisite Category: 64-bit]
   Controls the computational mode of the thread when an interrupt occurs that is directed to the guest supervisor state. At interrupt time, EPCR_{GICM} is copied into MSR_{CM} if the interrupt is directed to the guest supervisor state.
   0 Interrupts will execute in 32-bit mode.
   1 Interrupts will execute in 64-bit mode.

40 **Disable Guest TLB Management Instructions** (DGTMI)
   [Category: Embedded.Hypervisor]
   Controls whether guest supervisor state can execute any TLB management instructions.
   0 **tlbsr** and **tlbwe** (for a Logical to Real Address translation hit) are allowed to execute normally when MSR_{GS,PR} = 0b10.

41 **Disable MAS Interrupt Updates for Hypervisor** (DMIUH)
   [Category: Embedded.Hypervisor]
   Controls whether MAS registers are updated by hardware when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.
   0 MAS registers are set as described in Table 11 on page 1116 when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.
   1 MAS registers updates as described in Table 11 are disabled and MAS registers are left unchanged when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.

42 **Performance Monitor Interrupt Directed to Guest State** (PMGS)
   [Category: Embedded.Hypervisor]
   [Corequisite Category: Embedded.Performance Monitor]
   Controls whether a Performance Monitor Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor.
   0 Performance Monitor Interrupts that occur in the guest state are directed to the hypervisor state.
   1 Performance Monitor Interrupts that occur in the guest state are directed to the guest supervisor state.

43-63 Reserved
   This register is hypervisor privileged.

---

**Engineering Note**
EPCR only needs to be implemented if Category: Embedded.Hypervisor or category 64-bit are implemented.
4.3 Branch Facility Instructions

4.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The System Call instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

### System Call SC-form

<table>
<thead>
<tr>
<th>17</th>
<th>11</th>
<th>6</th>
<th>20</th>
<th>27</th>
<th>0</th>
<th>6</th>
<th>11</th>
<th>6</th>
<th>20</th>
<th>27</th>
<th>0</th>
</tr>
</thead>
</table>

sc LEV

[Category: Embedded.Hypervisor]

If LEV = 0 then

1. If MSRGS = 1 then
   - GSR0 ← iax CIA + 4
   - GSR1 ← MSR
   - If IVORs supported then
     - NIA ← GIVP0:47 || GIVOR848:59 || 0b0000
   - Else
     - NIA ← GIVP0:51 || 0x0120
     - MSR ← new_value (see below)

2. Else
   - SRR0 ← iax CIA + 4
   - SRR1 ← MSR
   - If IVORs supported then
     - NIA ← IVP0:47 || IVOR848:59 || 0b0000
   - Else
     - NIA ← IVP0:51 || 0x0120
     - MSR ← new_value (see below)

If LEV = 1 then

1. SRR0 ← iax CIA + 4
2. SRR1 ← MSR
3. If IVORs supported then
   - NIA ← IVP0:47 || IVOR4048:59 || 0b0000
4. Else
   - NIA ← IVP0:51 || 0x300
   - MSR ← new_value (see below)

If category E.HV is not implemented, the System Call instruction behaves as if MSRGS = 0 and LEV = 0.

If MSRGS = 0 or if LEV = 1, the effective address of the instruction following the System Call instruction is placed into SRR0 and the contents of the MSR are copied into SRR1. Otherwise, the effective address of the instruction following the System Call instruction is placed into GSRR0 and the contents of the MSR are copied into GSRR1.

If LEV=0, a System Call interrupt is generated. If LEV=1, an Embedded Hypervisor System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 7.6.10 and Section 7.6.30.

If LEV=0 and the thread is in guest state, the interrupt causes the next instruction to be fetched from the effective address based on one of the following:

- GIVP0:47||GIVOR848:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- GIVP0:51||0x120 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

If LEV=0 and the thread is in hypervisor state, the interrupt causes the next instruction to be fetched from the effective address based on one of the following:

- IVP0:47||IVOR848:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- IVP0:51||0x120 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

If LEV=1, the interrupt causes the next instruction to be fetched from the effective address based on one of the following:

- IVP0:47||IVOR4048:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- IVP0:51||0x300 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

This instruction is context synchronizing.

**Special Registers Altered:**

<table>
<thead>
<tr>
<th>SRR0</th>
<th>GSRR0</th>
<th>SRR1</th>
<th>GSRR1</th>
<th>MSR</th>
</tr>
</thead>
</table>

**Programming Note**

`sc` serves as both a basic and an extended mnemonic. The Assembler will recognize an `sc` mnemonic with one operand as the basic form, and an `sc` mnemonic with no operand as the extended form. In the extended form, the LEV operand is omitted and assumed to be 0.
Return From Interrupt  XL-form

rfi

<table>
<thead>
<tr>
<th>19</th>
<th>///</th>
<th>///</th>
<th>///</th>
<th>50</th>
</tr>
</thead>
</table>

MSR ← SRR1  
NIA ←lea SRR0:61 || 0b00

The *rfi* instruction is used to return from a base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of SRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0:61||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the applicable save/restore register 0 by the interrupt processing mechanism (see Section 7.6 on page 1161) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in SRR0 at the time of the execution of the *rfi*).

This instruction is privileged and context synchronizing.

Category: Embedded, Hypervisor

When *rfi* is executed in guest state, the instruction is mapped to *rfgi* and *rfgi* is executed instead.

Special Registers Altered:

MSR

Return From Critical Interrupt  XL-form

rfci

<table>
<thead>
<tr>
<th>19</th>
<th>///</th>
<th>///</th>
<th>///</th>
<th>51</th>
</tr>
</thead>
</table>

MSR ← CSRR1  
NIA ←lea CSRR0:61 || 0b00

The *rfci* instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of CSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address CSRR0:61||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or CSRR0 by the interrupt processing mechanism (see Section 7.6 on page 1161) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in CSRR0 at the time of the execution of the *rfci*).

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

MSR
Return From Debug Interrupt X-form

rfdi

[Category: Embedded.Enhanced Debug]

\[
\begin{array}{cccccccc}
\text{MSR} & \leftarrow & \text{DSRR1} \\
\text{NIA} & \leftarrow & \text{iea} & \text{DSRR0}^{0.61} & || & 0b00
\end{array}
\]

The `rfdi` instruction is used to return from a Debug interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address DSRR0^{0.61}||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, CSRR0, or DSRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in DSRR0 at the time of the execution of the `rfdi`).

This instruction is hypervisor privileged and context synchronizing.

**Special Registers Altered:**

- MSR

---

Return From Machine Check Interrupt XL-form

rfmci

\[
\begin{array}{cccccccc}
\text{MSR} & \leftarrow & \text{MCSRR1} \\
\text{NIA} & \leftarrow & \text{iea} & \text{MCSRR0}^{0.61} & || & 0b00
\end{array}
\]

The `rfmci` instruction is used to return from a Machine Check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address MCSRR0^{0.61}||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, CSRR0, MCSRR0, or DSRR0 by the interrupt processing mechanism (see Section 7.6 on page 1161) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in MCSRR0 at the time of the execution of the `rfmci`).

This instruction is hypervisor privileged and context synchronizing.

**Special Registers Altered:**

- MSR
Return From Guest Interrupt  

**Return From Guest Interrupt**  

```
rfgi  [Category:Embedded.Hypervisor]
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>102</td>
</tr>
</tbody>
</table>

The `rfgi` instruction is used to return from a guest state base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of Guest Save/Restore Register 1 are placed into the MSR. If the `rfgi` is executed in the guest supervisor state (MSRGS PR = 0b10), the bit MSR GS is not modified and the bits MSR UCLE DE PMM are modified only if the associated bits in the Machine State Register Protect (MSRP) Register are set to 0. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address GSRR0:61||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the associated save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in GSRR0 at the time of the execution of the `rfgi`).

This instruction is privileged and context synchronizing.

**Special Registers Altered:**

- MSR

Embedded Hypervisor Privilege  

**Embedded Hypervisor Privilege**  

```
ehpriv  OC  [Category: Embedded.Hypervisor]
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>270</td>
</tr>
</tbody>
</table>

The `ehpriv` instruction generates an Embedded Hypervisor Privilege Exception resulting in an Embedded Hypervisor Privilege Interrupt.

The OC field may be used by hypervisor software to provide a facility for emulated virtual instructions.

**Special Registers Altered:**

- None

---

**Programming Note**

The `ehpriv` instruction is analogous to a guaranteed illegal instruction encoding in that it guarantees that an Embedded Hypervisor Privilege exception is generated. The instruction is useful for programs that need to communicate information to the hypervisor software, particularly as a means for implementing breakpoint operations in a hypervisor managed debugger.

---

**Programming Note**

`ehpriv` serves as both a basic and an extended mnemonic. The Assembler will recognize an `ehpriv` mnemonic with one operand as the basic form, and an `ehpriv` mnemonic with no operand as the extended form. In the extended form, the OC operand is omitted and assumed to be 0.
Chapter 5. Fixed-Point Facility

5.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

5.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the `mfspr` (page 1054) and `mtspr` (page 1053) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

5.3 Fixed-Point Facility Registers

5.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the hardware. The contents of the PVR can be copied to a GPR by the `mfspr` instruction. Read access to the PVR is privileged; write access is not provided.

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations, such as which optional facilities and instructions are supported.</td>
</tr>
<tr>
<td>Revision</td>
<td>A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.</td>
</tr>
</tbody>
</table>

Version numbers are assigned by the Power ISA Architecture process. Revision numbers are assigned by an implementation-defined process.

5.3.2 Chip Information Register

The Chip Information Register (CIR) is a 32-bit read-only register that contains a value identifying the manufacturer and other characteristics of the chip on which the processor is implemented. The contents of the CIR can be copied to a GPR by the `mfspr` instruction. Read access to the CIR is privileged; write access is not provided.

![Figure 11. Chip Information Register](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:35</td>
<td>Manufacturer ID (ID) A four-bit field that identifies the manufacturer of the chip.</td>
</tr>
<tr>
<td>36:63</td>
<td>Implementation-dependent.</td>
</tr>
</tbody>
</table>

5.3.3 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be read using `mfspr` and written using `mtspr`. Read access to the PIR is privileged; write access, if provided, is hypervisor privileged.
Read accesses to the PIR in guest supervisor state are mapped to the GPIR. The means by which the PIR is initialized are implementation-dependent. The PIR can be used to identify the thread globally among all threads in a system that contains multiple threads. This facilitates more efficient usage of the Processor Control facility (see Section 11).

5.3.4 Guest Processor Identification Register [Category: Embedded.Hypervisor]

The Guest Processor Identification Register (GPIR) is a 32-bit register that contains a value that can be used to distinguish the thread from other threads in the system. The contents of the GPIR can be read using mfspr and written using mtspr. Read access to the GPIR is privileged; write access, if provided, is hypervisor privileged. Some previous implementations did not allow write access to the PIR. This allows the hypervisor to easily track changes to the GPIR. If write access is allowed to the PIR, write access should be allowed to the GPIR and should only be available to the hypervisor.

Programming Note

mfspr RT,PIR should be used to read GPIR in guest supervisor state. See Section 2.2.1, “Register Mapping”.

5.3.5 Program Priority Register

32-bit

Privileged programs may set a wider range of program priorities in the PRI field of PPR32 than may be set by problem-state programs (see Section 3.1 of Book II). Problem-state programs may only set values in the range of 0b010 to 0b100. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. If a program attempts to set a value that is not available to it, the PRI field remains unchanged. The values and their corresponding meanings are as follows.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>very low</td>
</tr>
<tr>
<td>010</td>
<td>low</td>
</tr>
<tr>
<td>011</td>
<td>medium low</td>
</tr>
<tr>
<td>100</td>
<td>medium</td>
</tr>
<tr>
<td>101</td>
<td>medium high</td>
</tr>
<tr>
<td>110</td>
<td>high</td>
</tr>
<tr>
<td>111</td>
<td>very high</td>
</tr>
</tbody>
</table>

5.3.6 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

<table>
<thead>
<tr>
<th>SPRG0</th>
<th>SPRG1</th>
<th>SPRG2</th>
<th>SPRG3</th>
<th>SPRG4</th>
<th>SPRG5</th>
<th>SPRG6</th>
<th>SPRG7</th>
<th>SPRG8</th>
</tr>
</thead>
</table>

0 through SPRG2

USPRG0 was made a 32-bit register and renamed to VRSAVE; see Sections 3.2.3 and 5.3.3 of Book I.
These 64-bit registers can be accessed only in supervisor mode.

[Category:Embedded.Hypervisor]
Access to these registers in guest supervisor state is mapped to GSPRG0 through GSPRG2.

SPRG3
This 64-bit register can be read in supervisor mode and can be written only in supervisor mode. It is implementation-dependent whether or not this register can be read in user mode.

[Category:Embedded.Hypervisor]
Access to this register in guest state is mapped to GSPRG3.

SPRG4 through SPRG7
These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor and user modes.

SPRG8 through SPRG9
These 64-bit registers can be accessed only in supervisor mode.

---

**Programming Note**
The intended use for SPRG9 is for internal debug exception handling.

---

GSPRG0 through GSPRG2

[Category:Embedded.Hypervisor]
These 64-bit registers can be accessed only in supervisor mode.

GSPRG3

[Category:Embedded.Hypervisor]
This 64-bit register can be read in supervisor mode and can be written only in supervisor mode. If an implementation permits problem state read access to SPRG3, the problem state read access is remapped to GSPRG3.

SPRGi or GSPRGi can be read using `mfspr` and written using `mtspr`.

---

**Programming Note**

`mfspr RT,SPRGi` should be used to read GSPRGi in guest state. `mtspr SPRGi,RS` should be used to write GSPRGi in guest state. See Section 2.2.1, “Register Mapping”.
5.3.7 External Process ID Registers [Category: Embedded.External PID]

The External Process ID Registers provide capabilities for loading and storing General Purpose Registers and performing cache management operations using a supplied context other than the context normally used by the programming model.

Two SPRs describe the context for loading and storing using external contexts. The External Process ID Load Context (EPLC) Register provides the context for External Process ID Load instructions, and the External Process ID Store Context (EPSC) Register provides the context for External Process ID Store instructions. Each of these registers contains a PR (privilege) bit, an AS (address space) bit, a Process ID, a GS (guest state) bit <E.HV>, and an LPID <E.HV>. Changes to the EPLC or the EPSC Register require that a context synchronizing operation be performed prior to using any External Process ID instructions that use these registers.

External Process ID instructions that use the context provided by the EPLC register include lbepx, lhepx, lwepx, ldepx, dcbtep, dcbfep, icbtep, Ifdepx, evlddepx, lvpx, and lvpxl and those that use the context provided by the EPSC register include stbepx, sthepx, stwepx, stdepx, dcbzep, stfdepx, evstddepx, stvepx, stvepxl, and dcbtstep. Instruction definitions appear in Section 5.4.3.

System software configures the EPLC register to reflect the Process ID, AS, PR, GS <E.HV>, and LPID <E.HV> state from the context that it wishes to perform loads from and configures the EPSC register to reflect the Process ID, AS, PR, GS <E.HV>, and LPID <E.HV> state from the context it wishes to perform stores to. Software then issues External Process ID instructions to manipulate data as required.

When the an External Process ID Load instruction is executed, it uses the context information in the EPLC Register instead of the normal context with respect to address translation and storage access control. EPLC<PR> is used in place of MSR<PR>, EPLC<AS> is used in place of MSR<AS>, EPLC<GS> is used in place of MSR<GS>, EPLC<LPID> is used in place of LPIDR <E.HV>, and EPLC<ELS> is used in place of LPIDR <E.HV>. Similarly, when the an External Process ID Store instruction is executed, it uses the context information in the EPSC Register instead of the normal context with respect to address translation and storage access control. EPSC<PR> is used in place of MSR<PR>, EPSC<AS> is used in place of MSR<AS>, EPSC<GS> is used in place of MSR<GS>, EPSC<ELS> is used in place of LPIDR <E.HV>, and EPSC<ELS> is used in place of LPIDR <E.HV>. Translation occurs using the new substituted values.

If the TLB lookup is successful, the storage access control mechanism grants or denies the access using context information from EPLC<PR> or EPSC<PR> for loads and stores respectively. If access is not granted, a Data Storage interrupt occurs, and the ESR<PID> bit is set to 1. If the operation was a Store, the ESR<ST> bit is also set to 1.

5.3.7.1 External Process ID Load Context (EPLC) Register

The EPLC register contains fields to provide the context for External Process ID Load instructions.

Figure 15. External Process ID Load Context Register

These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td><strong>External Load Context PR Bit</strong> (EPR)</td>
</tr>
<tr>
<td></td>
<td>Used in place of MSR&lt;PR&gt; by the storage access control mechanism when an External Process ID Load instruction is executed.</td>
</tr>
<tr>
<td>0</td>
<td>Supervisor mode</td>
</tr>
<tr>
<td>1</td>
<td>User mode</td>
</tr>
<tr>
<td>33</td>
<td><strong>External Load Context AS Bit</strong> (EAS)</td>
</tr>
<tr>
<td></td>
<td>Used in place of MSR&lt;AS&gt; for translation when an External Process ID Load instruction is executed, and, if this Load instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of MSR&lt;AS&gt;:</td>
</tr>
<tr>
<td>0</td>
<td>Address space 0</td>
</tr>
<tr>
<td>1</td>
<td>Address space 1</td>
</tr>
<tr>
<td>34</td>
<td><strong>External Load Context GS Bit</strong> (EGS)</td>
</tr>
<tr>
<td></td>
<td>[Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Used in place of MSR&lt;GS&gt; for translation when an External Process ID Load instruction is executed.</td>
</tr>
<tr>
<td>0</td>
<td>Hypervisor state</td>
</tr>
<tr>
<td>1</td>
<td>Guest state</td>
</tr>
</tbody>
</table>

**Programming Note**

When a mtsprr instruction is executed that targets EPLC, the EGS and ELPD fields are only modified if the thread is in hypervisor state.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Reserved</td>
</tr>
<tr>
<td>36:47</td>
<td><strong>External Load Context LPID Value</strong> (ELPID)</td>
</tr>
<tr>
<td></td>
<td>[Category: Embedded.Hypervisor]</td>
</tr>
<tr>
<td></td>
<td>Used in place of LPIDR register for translation</td>
</tr>
</tbody>
</table>
when an External Process ID Load instruction is executed.

**Programming Note**

When a `mtspr` instruction is executed that targets EPLC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>48:49</td>
<td>Reserved</td>
</tr>
<tr>
<td>50:63</td>
<td>External Load Context Process ID Value (EPID) Used in place of the Process ID register value for translation when an external Process ID Load instruction is executed, and, if this Load instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of PID contents.</td>
</tr>
</tbody>
</table>

### 5.3.7.2 External Process ID Store Context (EPSC) Register

The EPSC register contains fields to provide the context for External Process ID Store instructions. The field encoding is the same as the EPLC Register.

![Figure 16. External Process ID Store Context Register](image)

These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td><strong>External Store Context PR Bit</strong> (EPR) Used in place of MSRPR by the storage access control mechanism when an External Process ID Store instruction is executed. 0 Supervisor mode 1 User mode</td>
</tr>
<tr>
<td>33</td>
<td><strong>External Store Context AS Bit</strong> (EAS) Used in place of MSRD for translation when an External Process ID Store instruction is executed, and, if this Store instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of MSRD. 0 Address space 0 1 Address space 1</td>
</tr>
<tr>
<td>34</td>
<td><strong>External Store Context GS Bit</strong> (EGS) [Category: Embedded,Hypervisor] Used in place of MSRGS for translation when an External Process ID Store instruction is executed. 0 Hypervisor state 1 Guest state</td>
</tr>
</tbody>
</table>

**Programming Note**

When a `mtspr` instruction is executed that targets EPSC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>48:49</td>
<td>Reserved</td>
</tr>
<tr>
<td>50:63</td>
<td>External Store Context Process ID Value (EPID) Used in place of LPIDR register for translation when an External Process ID Store instruction is executed.</td>
</tr>
</tbody>
</table>

**Programming Note**

When a `mtspr` instruction is executed that targets EPSC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>48:49</td>
<td>Reserved</td>
</tr>
<tr>
<td>50:63</td>
<td><strong>External Store Context Process ID Value (EPID)</strong> Used in place of the Process ID register value for translation when an external PID Store instruction is executed, and, if this Store instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of PID contents.</td>
</tr>
</tbody>
</table>
5.4 Fixed-Point Facility Instructions

5.4.1 Move To/From System Register Instructions

The Move To Special Purpose Register and Move From Special Purpose Register instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in supervisor mode. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given below, the “defined” SPR numbers are the SPR numbers shown in Table 17 and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers.

Extended mnemonics

Extended mnemonics are provided for the \texttt{mtspr} and \texttt{mfspr} instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand; see Appendix B.

<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR\textsubscript{1}</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat\textsuperscript{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>spr\textsubscript{3,9}</td>
<td>spr\textsubscript{0,4}</td>
<td>\texttt{mtspr}</td>
<td>\texttt{mfspr}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000 00001</td>
<td>XER</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>00000 00011</td>
<td>DSCR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>00000 01000</td>
<td>LR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>00000 01001</td>
<td>CTR</td>
<td>no</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>17</td>
<td>00000 10001</td>
<td>DSCR</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>22</td>
<td>00000 10110</td>
<td>DEC</td>
<td>yes\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>26</td>
<td>00000 11010</td>
<td>SRR0</td>
<td>yes\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>27</td>
<td>00000 11011</td>
<td>SRR1</td>
<td>yes\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>48</td>
<td>00001 10000</td>
<td>PID</td>
<td>yes</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>53</td>
<td>00001 10101</td>
<td>GDECAR</td>
<td>hypv\textsuperscript{a}</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>54</td>
<td>00001 10110</td>
<td>DECAR</td>
<td>hypv\textsuperscript{a}</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>55</td>
<td>00001 10111</td>
<td>MCI\textsuperscript{3}PR</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>56</td>
<td>00001 11000</td>
<td>LPER</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>57</td>
<td>00001 11001</td>
<td>LPERU</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>58</td>
<td>00001 11010</td>
<td>CSRRO</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>59</td>
<td>00001 11011</td>
<td>CSRRI</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>60</td>
<td>00001 11100</td>
<td>GTS\textsuperscript{3}WR</td>
<td>hypv\textsuperscript{a}</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>61</td>
<td>00001 11101</td>
<td>DEAR</td>
<td>yes\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>62</td>
<td>00001 11110</td>
<td>ESR</td>
<td>yes\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>63</td>
<td>00001 11111</td>
<td>IVPR</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>01000 00000</td>
<td>VRSAVE</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>259</td>
<td>01000 00011</td>
<td>SPRG3</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>260-263</td>
<td>01000 001xx</td>
<td>SPRG[4-7]</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>268</td>
<td>01000 01000</td>
<td>TB</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>269</td>
<td>01000 01010</td>
<td>TBU</td>
<td>-</td>
<td>no</td>
<td>32\textsuperscript{s}</td>
</tr>
<tr>
<td>272-275</td>
<td>01000 100xx</td>
<td>SPRG[0-3]</td>
<td>yes\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>64</td>
</tr>
<tr>
<td>276-279</td>
<td>01000 101xx</td>
<td>SPRG[4-7]</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>282</td>
<td>01000 11010</td>
<td>EAR</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>284</td>
<td>01000 11100</td>
<td>TBL</td>
<td>hypv\textsuperscript{a}</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>283</td>
<td>01000 11111</td>
<td>CIR</td>
<td>-</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>285</td>
<td>01000 11111</td>
<td>TBU</td>
<td>hypv\textsuperscript{a}</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>286</td>
<td>01000 11110</td>
<td>PIR</td>
<td>hypv\textsuperscript{a}</td>
<td>yes\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>287</td>
<td>01000 11111</td>
<td>PVR</td>
<td>-</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>304</td>
<td>01001 10000</td>
<td>DBSR</td>
<td>hypv\textsuperscript{a,3}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>306</td>
<td>01001 10010</td>
<td>DBSRWR</td>
<td>hypv\textsuperscript{a}</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>307</td>
<td>01001 10011</td>
<td>EPCR</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>308</td>
<td>01001 10100</td>
<td>DBCR0</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>309</td>
<td>01001 10101</td>
<td>DBCR1</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
<tr>
<td>310</td>
<td>01001 10110</td>
<td>DBCR2</td>
<td>hypv\textsuperscript{a}</td>
<td>hypv\textsuperscript{a}</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 17. SPR Numbers (Sheet 1 of 4)
<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length</th>
<th>Cat&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>311</td>
<td>0001 10111</td>
<td>MSRP&lt;sup&gt;5&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>312</td>
<td>0001 11000</td>
<td>IAC1</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>313</td>
<td>0001 11001</td>
<td>IAC2</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>314</td>
<td>0001 11010</td>
<td>IAC3</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>315</td>
<td>0001 11011</td>
<td>IAC4</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>316</td>
<td>0001 11100</td>
<td>DAC1</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>317</td>
<td>0001 11101</td>
<td>DAC2</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>318</td>
<td>0010 10000</td>
<td>TSR</td>
<td>yes&lt;sup&gt;3&lt;/sup&gt;</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>319</td>
<td>0010 10010</td>
<td>LPIDR</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>320</td>
<td>0010 10011</td>
<td>MAS5</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>321</td>
<td>0010 10100</td>
<td>TCR</td>
<td>yes&lt;sup&gt;3&lt;/sup&gt;</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>322</td>
<td>0010 10101</td>
<td>MAS8</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>323</td>
<td>0010 10110</td>
<td>LRATCFG</td>
<td>-</td>
<td>32</td>
<td>E.HV.LRAT</td>
</tr>
<tr>
<td>324</td>
<td>0010 10111</td>
<td>LRATPS</td>
<td>-</td>
<td>32</td>
<td>E.HV.LRAT</td>
</tr>
<tr>
<td>325</td>
<td>0010 11011</td>
<td>MAS8IMAS6</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E.HV:64</td>
</tr>
<tr>
<td>326</td>
<td>0010 11101</td>
<td>MAS8IMAS1</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E.HV:64</td>
</tr>
<tr>
<td>327</td>
<td>0010 11110</td>
<td>EPTCFG</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.PT</td>
</tr>
<tr>
<td>328</td>
<td>0010 11111</td>
<td>GSPPRGO-3</td>
<td>yes</td>
<td>64</td>
<td>E.HV</td>
</tr>
<tr>
<td>329</td>
<td>0011 10010</td>
<td>MAS7IMAS3</td>
<td>yes</td>
<td>64</td>
<td>E:64</td>
</tr>
<tr>
<td>330</td>
<td>0011 10011</td>
<td>MAS0IMAS1</td>
<td>yes</td>
<td>64</td>
<td>E:64</td>
</tr>
<tr>
<td>331</td>
<td>0011 10110</td>
<td>GDEC</td>
<td>yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>332</td>
<td>0011 10111</td>
<td>GTCH</td>
<td>yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>333</td>
<td>0011 10110</td>
<td>GTSR</td>
<td>yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>334</td>
<td>0011 11010</td>
<td>GSRH0</td>
<td>yes</td>
<td>64</td>
<td>E.HV</td>
</tr>
<tr>
<td>335</td>
<td>0011 11110</td>
<td>GSRH1</td>
<td>yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>336</td>
<td>0011 11111</td>
<td>GEPR</td>
<td>yes</td>
<td>64</td>
<td>E.HV:EXP</td>
</tr>
<tr>
<td>337</td>
<td>0011 11110</td>
<td>GDEAR</td>
<td>yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>338</td>
<td>0011 11111</td>
<td>GESR</td>
<td>yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>339</td>
<td>0011 11100</td>
<td>IVOR0-15</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>340</td>
<td>0011 11110</td>
<td>IVOR38-41</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>341</td>
<td>0011 10100</td>
<td>IVOR42</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV.LRAT</td>
</tr>
<tr>
<td>342</td>
<td>0011 10101</td>
<td>TENS</td>
<td>-</td>
<td>64</td>
<td>E.MT</td>
</tr>
<tr>
<td>343</td>
<td>0011 10110</td>
<td>TENS</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E.MT</td>
</tr>
<tr>
<td>344</td>
<td>0011 10111</td>
<td>TENC</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E.MT</td>
</tr>
<tr>
<td>345</td>
<td>0011 10100</td>
<td>IVOR2-3</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>346</td>
<td>0011 10111</td>
<td>IVOR4</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>347</td>
<td>0011 11010</td>
<td>IVOR8</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>348</td>
<td>0011 11110</td>
<td>IVOR13</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>349</td>
<td>0011 11111</td>
<td>IVOR14</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>350</td>
<td>0011 11111</td>
<td>TiR</td>
<td>-</td>
<td>64</td>
<td>E.MT</td>
</tr>
<tr>
<td>351</td>
<td>0011 11111</td>
<td>GIVPR</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>64</td>
<td>E.HV</td>
</tr>
<tr>
<td>352</td>
<td>0011 01110</td>
<td>GIVOR10</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>353</td>
<td>0011 01111</td>
<td>GIVOR11</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>354</td>
<td>0011 01111</td>
<td>GIVOR12</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>355</td>
<td>0010 01110</td>
<td>SPEFSCR</td>
<td>no</td>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>356</td>
<td>0010 01111</td>
<td>ATB/ATBL</td>
<td>-</td>
<td>64</td>
<td>ATB</td>
</tr>
<tr>
<td>357</td>
<td>0010 01111</td>
<td>ATBU</td>
<td>-</td>
<td>32</td>
<td>ATB</td>
</tr>
<tr>
<td>358</td>
<td>0010 10000</td>
<td>IVOR32</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>359</td>
<td>0010 10001</td>
<td>IVOR33</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>360</td>
<td>0010 10010</td>
<td>IVOR34</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>361</td>
<td>0010 10011</td>
<td>IVOR35</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.PM</td>
</tr>
<tr>
<td>362</td>
<td>0010 10100</td>
<td>IVOR36</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.PC</td>
</tr>
<tr>
<td>363</td>
<td>0010 10101</td>
<td>IVOR37</td>
<td>hypv&lt;sup&gt;5&lt;/sup&gt;</td>
<td>32</td>
<td>E.PC</td>
</tr>
</tbody>
</table>

Figure 17. SPR Numbers (Sheet 2 of 4)
<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR\textsuperscript{1}</th>
<th>Register Name</th>
<th>Privileged Length</th>
<th>Length</th>
<th>Cat\textsuperscript{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>570</td>
<td>spr\textsubscript{g} 11010</td>
<td>MCSR0</td>
<td>hypv\textsuperscript{g} 64</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>571</td>
<td>spr\textsubscript{g} 11011</td>
<td>MCSR1</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>572</td>
<td>spr\textsubscript{0,4} 11100</td>
<td>MCSR</td>
<td>hypv\textsuperscript{g} 64</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>574</td>
<td>spr\textsubscript{0,4} 11110</td>
<td>DSRR0</td>
<td>yes yes 64</td>
<td>E,ED</td>
<td></td>
</tr>
<tr>
<td>575</td>
<td>spr\textsubscript{0,4} 11111</td>
<td>DSRR1</td>
<td>yes yes 32</td>
<td>E,ED</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>spr\textsubscript{0,4} 11100</td>
<td>SPRG8</td>
<td>hypv\textsuperscript{g} 64</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>605</td>
<td>spr\textsubscript{0,4} 11110</td>
<td>SPRG9</td>
<td>yes yes 64</td>
<td>E,ED</td>
<td></td>
</tr>
<tr>
<td>624</td>
<td>spr\textsubscript{0,4} 10000</td>
<td>MAS0</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>625</td>
<td>spr\textsubscript{0,4} 10001</td>
<td>MAS1</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>626</td>
<td>spr\textsubscript{0,4} 10010</td>
<td>MAS2</td>
<td>yes yes 64</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>627</td>
<td>spr\textsubscript{0,4} 10011</td>
<td>MAS3</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>628</td>
<td>spr\textsubscript{0,4} 10100</td>
<td>MAS4</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>630</td>
<td>spr\textsubscript{0,4} 10110</td>
<td>MAS6</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>631</td>
<td>spr\textsubscript{0,4} 10111</td>
<td>MAS2U</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>688-691</td>
<td>10101 100\text{xx}</td>
<td>TLB[0-3]CFG</td>
<td>- hypv\textsuperscript{g} 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>702</td>
<td>10101 11110</td>
<td>EPR</td>
<td>- yes\textsuperscript{g} 32</td>
<td>EXP</td>
<td></td>
</tr>
<tr>
<td>808</td>
<td>11001 01000</td>
<td>reserved\textsuperscript{10}</td>
<td>no no na</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>809</td>
<td>11001 01001</td>
<td>reserved\textsuperscript{10}</td>
<td>no no na</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>810</td>
<td>11001 01010</td>
<td>reserved\textsuperscript{10}</td>
<td>no no na</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>811</td>
<td>11001 01011</td>
<td>reserved\textsuperscript{10}</td>
<td>no no na</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>898</td>
<td>11100 00010</td>
<td>PPR32</td>
<td>no no 32</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>924</td>
<td>11100 11100</td>
<td>DCDBTRL\textsuperscript{5}</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E,CD</td>
<td></td>
</tr>
<tr>
<td>925</td>
<td>11100 11101</td>
<td>DCDBTRH\textsuperscript{5}</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E,CD</td>
<td></td>
</tr>
<tr>
<td>926</td>
<td>11100 11110</td>
<td>ICDBTRL\textsuperscript{6}</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E,CD</td>
<td></td>
</tr>
<tr>
<td>927</td>
<td>11100 11111</td>
<td>ICDBTRH\textsuperscript{6}</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E,CD</td>
<td></td>
</tr>
<tr>
<td>944</td>
<td>11101 10000</td>
<td>MAS7</td>
<td>yes yes 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>947</td>
<td>11101 10010</td>
<td>EPSC</td>
<td>yes yes 32</td>
<td>E,PD</td>
<td></td>
</tr>
<tr>
<td>948</td>
<td>11101 10100</td>
<td>EPLC</td>
<td>yes yes 32</td>
<td>E,PD</td>
<td></td>
</tr>
<tr>
<td>979</td>
<td>11110 10011</td>
<td>ICDBDR\textsuperscript{5}</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E,CD</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>11111 10100</td>
<td>MMUCSR0\textsuperscript{8}</td>
<td>hypv\textsuperscript{g} 32</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>1015</td>
<td>11111 10111</td>
<td>MMUCFG</td>
<td>- hypv\textsuperscript{g} 32</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

- This register is not defined for this instruction.
1 See Section 1.3.5 of Book I. If multiple categories are listed separated by a semicolon, all the listed categories must be implemented in order for the other columns of the line to apply. A comma separates two alternatives, and takes precedence over a semicolon; e.g., the EPCR (E.HV,E;64) must be implemented if either (a) category E.HV is implemented or (b) the implementation is Embedded and supports the 64-bit category.
2 This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E).
3 This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state; see Chapter 2 of Book III-E. Otherwise, the register is privileged.
4 The register can be written by the \texttt{dcread} instruction.
5 The register can be written by the \texttt{icread} instruction.
6 The register is Category: Phased-in.
7 If the Embedded.Hypervisor category is supported, this register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E). Otherwise, the register is privileged.
8 The register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state, and guest references to the register are redirected to the corresponding guest register (see Chapter 2 of Book III-E). Otherwise the register is privileged.
9 Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.

Figure 17. SPR Numbers (Sheet 3 of 4)
Move To Special Purpose Register
XFX-form

\[
\text{mtspr SPR,RS}
\]

\[
\begin{array}{ccccccc}
31 & 6 & 11 & \text{spr} & 21 & 467 & 7 \\
\hline
\end{array}
\]

\[
n \leftarrow \text{spr}_{5:9} | | \text{spr}_{0:4}
\]

\[
\text{switch (n)}
\begin{align*}
\text{case (808, 809, 810, 811):} & \\
\text{default:} & \\
\text{if length(SPR(n)) = 64 then} & \\
\text{SPR(n) } \leftarrow (\text{RS}) & \\
\text{else} & \\
\text{SPR(n) } \leftarrow (\text{RS})_{32:63}
\end{align*}
\]

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

\[\text{spr}_0=1\] if and only if writing the register is privileged. Execution of this instruction specifying a defined and privileged register when MSRPR=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if \[\text{spr}_0=0\]: boundedly undefined results
- if \[\text{spr}_0=1\]:
  - if MSRPR=1: Privileged Instruction type Program interrupt; if MSRPR=0: boundedly undefined results

If the SPR number is set to a value that is shown in Figure 17 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

Special Registers Altered:
See Figure 17

---

**Compiler and Assembler Note**

For the \text{mtspr} and \text{mfspr} instructions, the SPR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.

**Programming Note**

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 12, “Synchronization Requirements for Context Alterations” on page 1235.
Move From Special Purpose Register
XFX-form

mfspr RT,SPR

\[ n \leftarrow spr_{5:9} || spr_{0:4} \]
\[
\text{switch } (n) \\
\text{case(808, 809, 810, 811):} \\
\text{default:} \\
\text{if length(SPR(n)) = 64 then} \\
\text{RT } \leftarrow \text{SPR(n)} \\
\text{else} \\
\text{RT } \leftarrow 32 \quad || \quad \text{SPR(n)}
\]

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

spr0=1 if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when MSRPR=1 causes a Privileged Instruction type Program interrupt.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 16 | 15 | 14 | 13 | 6 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| RT | spr | 339 | / |

Move To Device Control Register
XFX-form

mtdcr DCRN,RS

\[ \text{DCRN } \leftarrow \text{dcr}_{5:9} || \text{dcr}_{0:4} \]
\[ \text{DCR(DCRN) } \leftarrow (RS) \]

Let DCRN denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of register RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of (RS) are placed into the Device Control Register.

This instruction is privileged.

Special Registers Altered:
   Implementation-dependent.

Move To Device Control Register Indexed
X-form

mtdcrx RA,RS

\[ \text{mtdcrx 105,r2} \]
\[ \text{mtdcrux r1,r2} \]

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers supported are implementation-dependent.)

The contents of register RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of (RS) are placed into the Device Control Register.

The specification of Device Control Registers using mtdcrx, mtdcrux (see Book I), and mtdcr is implementation-dependent. For example, mtdcr 105,r2 and mtdcrx 1,r2 (where register r1 contains the value 105) may not produce identical results on an implementation.

This instruction is privileged.

Special Registers Altered:
   Implementation-dependent.
Move From Device Control Register
XFX-form

\[ \text{mfdcr} \quad \text{RT,DCRN} \]

[Category: Embedded.Device Control]

\[
\begin{array}{cccccc}
31 & \text{RT} & 21 &  \text{dcr} & 323 & / \\
0 & 6 & 11 & & 21 & 31
\end{array}
\]

\[
\text{DCRN} \leftarrow \text{dcr}_{5:9} \parallel \text{dcr}_{0:4}
\]

RT \leftarrow \text{DCR(DCRN)}

Let DCRN denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into register RT. For 32-bit Device Control Registers, the contents of the Device Control Register are placed into bits 32:63 of RT. Bits 0:31 of RT are set to 0.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move From Device Control Register
Indexed X-form

\[ \text{mfdr} \quad \text{RT,RA} \]

[Category: Embedded.Device Control]

\[
\begin{array}{ccccccc}
31 & \text{RT} & \text{RA} & \text{dcr} & 259 & / \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\text{DCRN} \leftarrow \text{(RA)}
\]

RT \leftarrow \text{DCR(DCRN)}

Let the contents of register RA denote a Device Control Register (the supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into register RT. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT. Bits 0:31 of RT are set to 0.

The specification of Device Control Registers using \text{mfdcr}x and \text{mfdcr}ux (see Book I) compared to the specification of Device Control Registers using \text{mfdcr} is implementation-dependent. For example, \text{mfdcr} r2,105 and \text{mfdcr}x r2,r1 (where register r1 contains the value 105) may not produce identical results on an implementation or between implementations. Also, accessing privileged Device Control Registers in supervisor mode with \text{mfdcr}ux is implementation-dependent.

This instruction is privileged.

Move To Machine State Register
X-form

\[ \text{mtmsr} \quad \text{RS} \]

\[
\begin{array}{cccccc}
31 & \text{RS} & 16 & 21 & 146 & / \\
0 & 6 & 11 & 16 & 21 & 31
\end{array}
\]

\[
\text{newmsr} \leftarrow \text{(RS)}_{32:63}
\]

if MSR_{CM} = 0 \& newmsr_{CM} = 1 then \text{NIA}_{0:31} \leftarrow 0

if MSR_{GS} = 1 then

\[
\begin{array}{l}
\text{newmsr}_{WE} \leftarrow \text{MSR}_{GS} \text{ WE} \\
\text{prots}_{0:31} \leftarrow 0 \\
\text{prots}_{UCLEP \text{ DEP PMMP}} \leftarrow \text{MSR}_{UCLEP \text{ DEP PMMP}} \\
\text{newmsr} \leftarrow \text{prots} \& \text{MSR} \mid \text{-prots} \& \text{newmsr} \\
\text{MSR} \leftarrow \text{newmsr}
\end{array}
\]

The contents of register RS_{32:63} are placed into the MSR. If the thread is changing from 32-bit mode to 64-bit mode, the next instruction is fetched from 320||\text{NIA}_{32:63}.

This instruction is privileged and execution synchronizing.

In addition, alterations to the EE or CE bits are effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing an \text{mtmsr} that sets MSR_{EE} to 1 will cause the External interrupt to be taken before the next instruction is executed, if no higher priority exception exists. Likewise, if MSR_{CE}=0 and a Critical Input interrupt is pending, executing an \text{mtmsr} that sets MSR_{CE} to 1 will cause the Critical Input interrupt to be taken before the next instruction is executed if no higher priority exception exists. (See Section 7.6 on page 1161.)

[Category: Embedded.Hypervisor]
GS, WE and bits protected with MSRP are only modified if \text{mtmsr} is executed in hypervisor state.

Special Registers Altered:
MSR

Programming Note
For a discussion of software synchronization requirements when altering certain MSR bits please refer to Chapter 12.
Move From Machine State Register X-form

**mfmsr RT**

\[
\begin{array}{cccccc}
& 31 & 6 & 11 & 16 & 21 & 83 & / \\
0 & & & & & & & \\
\end{array}
\]

\[
RT \leftarrow 32 \parallel MSR
\]

The contents of the MSR are placed into bits 32:63 of register RT and bits 0:31 of RT are set to 0.

This instruction is privileged.

**Special Registers Altered:**

None

---

Write MSR External Enable X-form

**wrtee RS**

\[
\begin{array}{cccccc}
& 31 & 6 & 11 & 16 & 21 & 131 & / \\
0 & & & & & & & \\
\end{array}
\]

\[
MSR_{EE} \leftarrow (RS)_{48}
\]

The content of (RS)_{48} is placed into MSR_{EE}.

Alteration of the MSR_{EE} bit is effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing a wrtee instruction that sets MSR_{EE} to 1 will cause the External interrupt to occur before the next instruction is executed, if no higher priority exception exists (Section 7.9, “Exception Priorities” on page 1190).

This instruction is privileged.

**Special Registers Altered:**

MSR
Write MSR External Enable Immediate
X-form

wrteei E

MSR_EE ← E

The value specified in the E field is placed into MSR_EE.

Alteration of the MSR_EE bit is effective as soon as the
instruction completes. Thus if MSR_EE=0 and an External
interrupt is pending, executing a wrtee instruction
that sets MSR_EE to 1 will cause the External interrupt to
occur before the next instruction is executed, if no
higher priority exception exists (Section 7.9, “Exception
Priorities” on page 1190).

This instruction is privileged.

Special Registers Altered:
MSR

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>17</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programming Note

wrtee and wrteei are used to provide atomic
update of MSR_EE. Typical usage is:

```
mfmsr Rn  #save EE in (Rn) 48
wrteei 0   #turn off EE
mfmsr Rn  #save EE in (Rn) 48
wrteei 0   #turn off EE
:    :     :
:    :     :
wrtee Rn  #code with EE disabled
          #restore EE without altering
          #other MSR bits that might
          #have changed
```
5.4.2 OR Instruction

or Rx,Rx,Rx can be used to set PPRPRI (see Figure 2 in Section 3.1 of Book II) as shown in Figure 18. PPRPRI remains unchanged if the privilege state of the thread executing the instruction is lower than the privilege indicated in the figure. (The encodings available to problem-state programs, as well as encodings for additional shared resource hints not shown here, are described in Section 3.2 of Book II.)

<table>
<thead>
<tr>
<th>Rx</th>
<th>PPR32PRI</th>
<th>Priority</th>
<th>Privileged</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>001</td>
<td>very low</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>low</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>011</td>
<td>medium low</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>medium</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>medium high</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>high</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>very high</td>
<td>hypv(^1)</td>
</tr>
</tbody>
</table>

\(^1\) If the Embedded.Hypervisor category is supported, this value is hypervisor privileged. Otherwise, the value is privileged.

Figure 18. Priority levels for or Rx,Rx,Rx
5.4.3 External Process ID Instructions [Category: Embedded.External PID]

*External Process ID* instructions provide capabilities for loading and storing General Purpose Registers and performing cache management operations using a supplied context other than the context normally used by translation.

The EPLC and EPSC registers provide external contexts for performing loads and stores. The EPLC and the EPSC registers are described in Section 5.3.7.

If an Alignment interrupt, Data Storage interrupt, or a Data TLB Error interrupt, occurs while attempting to execute an *External Process ID* instruction, ESR_{EPI}D is set to 1 indicating that the instruction causing the interrupt was an *External Process ID* instruction; any other applicable ESR bits are also set.

---

**Load Byte by External Process ID Indexed X-form**

\[
\text{lbepx} \quad \text{RT,RA,RB}
\]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RT | RA | RB | 95 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← \text{MEM}(EA,1)

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into RT_{56:63}. RT_{0:55} are set to 0.

For *lbepx*, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC_{EPR} is used in place of MSR_{PR}
- EPLC_{EAS} is used in place of MSR_{DS}
- EPLC_{EPI}D is used in place of PID
- EPLC_{EGS} is used in pace of MSR_{GS} <E.HV>
- EPLC_{ELPID} is used in pace of LPIDR <E.HV>

This instruction is privileged.

**Special Registers Altered:**
None

**Programming Note**
This instruction behaves identically to a *lbzx* instruction except for using the EPLC register to provide the translation context.

---

**Load Halfword by External Process ID Indexed X-form**

\[
\text{lhepx} \quad \text{RT,RA,RB}
\]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RT | RA | RB | 287 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← \text{MEM}(EA,2)

Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into RT_{48:63}. RT_{0:47} are set to 0.

For *lhepx*, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC_{EPR} is used in place of MSR_{PR}
- EPLC_{EAS} is used in place of MSR_{DS}
- EPLC_{EPI}D is used in place of PID
- EPLC_{EGS} is used in pace of MSR_{GS} <E.HV>
- EPLC_{ELPID} is used in pace of LPIDR <E.HV>

This instruction is privileged.

**Special Registers Altered:**
None

**Programming Note**
This instruction behaves identically to a *lhzx* instruction except for using the EPLC register to provide the translation context.
Load Word by External Process ID
Indexed X-form

lwepx  RT,RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← 32:0 | MEM(EA, 4)

Let the effective address (EA) be the sum (RA|0)+(RB). The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

For lwepx, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRS
- EPLCEPID is used in place of PID
- EPLCEEGS is used in place of MSRGS <E.HV>
- EPLCELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:
None

Programming Note
This instruction behaves identically to a lwzx instruction except for using the EPLC register to provide the translation context.

Load Doubleword by External Process ID
Indexed X-form

ldepx  RT,RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

For ldepx, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRS
- EPLCEPID is used in place of PID
- EPLCEEGS is used in place of MSRGS <E.HV>
- EPLCELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

Corequisite Categories:
64-Bit

Special Registers Altered:
None

Programming Note
This instruction behaves identically to a ldx instruction except for using the EPLC register to provide the translation context.
Store Byte by External Process ID
Indexed X-form

stbepx RS,RA,RB

\[
\begin{array}{cccc|c}
31 & RS & RA & RB & 223 \\
\end{array}
\]

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,1) ← (RS)_{56:63}

Let the effective address (EA) be the sum (RA|0)+(RB). (RS)_{56:63} are stored into the byte in storage addressed by EA.

For \textit{stbepx}, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:
EPSC\textsubscript{EPR} is used in place of MSR\textsubscript{PR}
EPSC\textsubscript{EAS} is used in place of MSR\textsubscript{DS}
EPSC\textsubscript{EPID} is used in place of PID
EPSC\textsubscript{EGS} is used in place of MSR\textsubscript{GS}<E.HV>
EPSC\textsubscript{ELPID} is used in place of LPIDR<E.HV>

This instruction is privileged.

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a \textit{stbx} instruction except for using the EPSC register to provide the translation context.

Store Halfword by External Process ID
Indexed X-form

sthexp RS,RA,RB

\[
\begin{array}{cccc|c}
31 & RS & RA & RB & 415 \\
\end{array}
\]

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)_{48:63}

Let the effective address (EA) be the sum (RA|0)+(RB). (RS)_{48:63} are stored into the halfword in storage addressed by EA.

For \textit{sthexp}, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:
EPSC\textsubscript{EPR} is used in place of MSR\textsubscript{PR}
EPSC\textsubscript{EAS} is used in place of MSR\textsubscript{DS}
EPSC\textsubscript{EPID} is used in place of PID
EPSC\textsubscript{EGS} is used in place of MSR\textsubscript{GS}<E.HV>
EPSC\textsubscript{ELPID} is used in place of LPIDR<E.HV>

This instruction is privileged.

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a \textit{sthx} instruction except for using the EPSC register to provide the translation context.
Store Word by External Process ID  
Indexed X-form

```
storex  RS,RA,RB

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)_{32:63}
```

Let the effective address (EA) be the sum (RA|0)+(RB). 
(RS)_{32:63} are stored into the word in storage addressed 
by EA.

For `storex`, the normal translation mechanism is not 
used. The contents of the EPSC register are used to 
provide the context in which translation occurs. The fol-
lowing substitutions are made for just the translation 
and access control process:

- EPSCPR is used in place of MSRPR
- EPSCAS is used in place of MSRDS
- EPSCPID is used in place of PID
- EPSCGS is used in pace of MSRGS <E.HV>
- EPSCLPID is used in pace of LPIDR <E.HV>

This instruction is privileged.

**Special Registers Altered:**
None

---

**Programming Note**

This instruction behaves identically to a `stwx` 
instruction except for using the EPSC register to 
provide the translation context.

---

Store Doubleword by External Process ID  
Indexed X-form

```
storepx  RS,RA,RB

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA,8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+(RB). 
(RS) is stored into the doubleword in storage 
addressed by EA.

For `storepx`, the normal translation mechanism is not 
used. The contents of the EPSC register are used to 
provide the context in which translation occurs. The fol-
lowing substitutions are made for just the translation 
and access control process:

- EPSCPR is used in place of MSRPR
- EPSCAS is used in place of MSRDS
- EPSCPID is used in place of PID
- EPSCGS is used in pace of MSRGS <E.HV>
- EPSCLPID is used in pace of LPIDR <E.HV>

This instruction is privileged.

**Corequisite Categories:**
64-Bit

**Special Registers Altered:**
None

---

**Programming Note**

This instruction behaves identically to a `stdx` 
instruction except for using the EPSC register to 
provide the translation context.
Data Cache Block Store by External PID

X-form

```
dcbstep RA,RB
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required, a block containing the byte addressed by EA is in the data cache of any thread, and any locations in the block are considered to be modified there, then those locations are written to main storage. Additional locations in the block may be written to main storage. The block ceases to be considered modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this thread, and any locations in the block are considered to be modified there, those locations are written to main storage. Additional locations in the block may be written to main storage, and the block ceases to be considered modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

The instruction is treated as a Load with respect to translation, memory protection, and is treated as a Write with respect to debug events.

This instruction is privileged.

For dcbstep, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRDS
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSR[GS]<E.HV>
- EPLCELPI is used in place of LPIDR<E.HV>

Special Registers Altered:
None

Programming Note
This instruction behaves identically to a dcbst instruction except for using the EPLC register to provide the translation context.

---

Data Cache Block Touch by External PID

X-form

```
dcbtep TH,RA,RB
```

Let the effective address (EA) be the sum (RA|0)+(RB).

The dcbtep instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA. If the Cache Specification category is supported, the nature of the hint is affected by TH values of 0b00000 to 0b00111. Values associated with the Stream category are ignored. See Section 4.3.2 of Book II for more information.

If the block is in a storage location that is Caching Inhibited or Guarded, the hint is ignored.

The only operation that is “caused” by the dcbtep instruction is the providing of the hint. The actions (if any) taken in response to the hint are not considered to be “caused by” or “associated with” the dcbtep instruction (e.g., dcbtep is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a sync instruction.

The dcbtep instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified in the dcbt instruction in Section 4.3.2 of Book II.

The instruction is treated as a Load, except that no interrupt occurs if a protection violation occurs.

The instruction is privileged.

The normal address translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRDS
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSR[GS]<E.HV>
- EPLCELPI is used in place of LPIDR<E.HV>

Special Registers Altered:
None

Extended Mnemonics:
Extended mnemonics are provided for the Data Cache Block Touch by External PID instruction so that it can...
be coded with the TH value as the last operand for all categories.

Extended: dcbtctep RA,RB,TH
Equivalent to:
dcbtep for TH values of 0b0000 - 0b0111;
other TH values are invalid.
dcbtdsep RA,RB,TH
dcbtep for TH values of 0b0000 or 0b1000 - 0b1010;
other TH values are invalid.

Programming Note
This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Data Cache Block Flush by External PID
X-form

dcbfep RA,RB,L

Let the effective address (EA) be the sum (RA|0)+(RB).

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 ("dcbf local") [Category: Embedded.Phased-In]

The L=1 form of the dcbfep instruction permits a program to limit the scope of the "flush" operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 ("dcbf local primary") [Category: Embedded.Phased-In]

The L=3 form of the dcbfep instruction permits a program to limit the scope of the "flush" operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

For the L operand, the value 2 is reserved. The results of executing a dcbfep instruction with L=2 are boundedly undefined.

Engineering Note

dcbfep with an L value of 2 should be treated as if the value were 0.

Extended: Equivalent to:
dcbtep for TH values of 0b0000 - 0b0111;
other TH values are invalid.
dcbtep for TH values of 0b0000 or 0b1000 - 0b1010;
other TH values are invalid.

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programmable Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Programming Note

This instruction behaves identically to a dcbt instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.
The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

The instruction is treated as a Load with respect to translation, memory protection, and is treated as a Write with respect to debug events.

This instruction is privileged.

The normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRDS
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSRGS<E.HV>
- EPLCELPID is used in place of LPIDR<E.HV>

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a dcbf instruction except for using the EPLC register to provide the translation context.

Extended Mnemonics:

Extended mnemonics are provided for the Data Cache Block Flush by External PID instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Section B.2 of Book III-E. The extended mnemonics are shown below.

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcbfep RA,RB</td>
<td>dcbfep RA,RB,0</td>
</tr>
<tr>
<td>dcbflep RA,RB</td>
<td>dcbfep RA,RB,1</td>
</tr>
<tr>
<td>dcbflep RA,RB</td>
<td>dcbfep RA,RB,3</td>
</tr>
</tbody>
</table>

Except in the dcbfep instruction description in this section, references to “dcbfep” in Books I-III imply L=0 unless otherwise stated or obvious from context; “dcbflep” is used for L=1 and “dcbflep” is used for L=3.

Programming Note

dcbfep serves as both a basic and an extended mnemonic. The Assembler will recognize a dcbfep mnemonic with three operands as the basic form, and a dcbfep mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.
Data Cache Block Touch for Store by External PID

The `dcbtstep` instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA. If the Cache Specification category is supported, the nature of the hint is affected by TH values of 0b00000 to 0b00111. Values associated with the Stream category are ignored. See Section 4.3.2 of Book II for more information.

If the block is in a storage location that is Caching Inhibited or Guarded, the hint is ignored.

The only operation that is “caused” by the `dcbtstep` instruction is the providing of the hint. The actions (if any) taken in response to the hint are not considered to be “caused by” or “associated with” the `dcbtstep` instruction (e.g., `dcbtstep` is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a `sync` instruction.

The `dcbtstep` instruction may complete before the operation it causes has been performed.

The instruction is treated as a Store, except that no interrupt occurs if a protection violation occurs.

The instruction is privileged.

The normal address translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- `EPLCEPR` is used in place of `MSRPR`
- `EPLCEAS` is used in place of `MSRDS`
- `EPLCEPID` is used in place of `PID`
- `EPLCEGSS` is used in place of `MSR[GS] <E.HV>`
- `EPLCELPID` is used in place of `LPIDR <E.HV>`

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the `Data Cache Block Touch for Store by External PID` instruction so that it can be coded with the TH value as the last operand for all categories.

**Extended:**

- `dcbtstep RA,RB,TH`

**Equivalent to:**

- `dcbtstep` for TH values of 0b0000 - 0b0111;
  - other TH values are invalid.

### Programming Note

This instruction behaves identically to a `dcbtst` instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

---

<table>
<thead>
<tr>
<th>dcbtstep</th>
<th>TH, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>TH</td>
</tr>
</tbody>
</table>
**Instruction Cache Block Invalidate by External PID X-form**

<table>
<thead>
<tr>
<th>icbiep</th>
<th>RA, RB</th>
</tr>
</thead>
</table>

Let the effective address (EA) be the sum \((\text{RA}|0)+(\text{RB})\).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any thread, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of this thread, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

The instruction is treated as a **Load**.

This instruction is privileged.

For **icbiep**, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRD\(S\)
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSR[GS] <E.HV>
- EPLCELPID is used in place of LPIDR <E.HV>

**Special Registers Altered:**

None

---

**Programming Note**

This instruction behaves identically to an **icbi** instruction except for using the EPLC register to provide the translation context.

---

**Data Cache Block set to Zero by External PID X-form**

<table>
<thead>
<tr>
<th>dcbzep</th>
<th>RA, RB</th>
</tr>
</thead>
</table>

Let the effective address (EA) be the sum \((\text{RA}|0)+(\text{RB})\).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a **Store**.

This instruction is privileged.

The normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC\(EPR\) is used in place of MSR\(PR\)
- EPSC\(EAS\) is used in place of MSR\(DS\)
- EPSC\(EPID\) is used in place of PID
- EPSC\(EGS\) is used in place of MSR[GS] <E.HV>
- EPSC\(ELPID\) is used in place of LPIDR <E.HV>

**Special Registers Altered:**

None

---

**Programming Note**

See the Programming Notes for the **dcbz** instruction.

---

**Programming Note**

This instruction behaves identically to a **dcbz** instruction except for using the EPSC register to provide the translation context.
Load Floating-Point Double by External Process ID Indexed X-form

\text{lfdepx} \ FRT,RA,RB

\begin{array}{ccc}
31 & FRT & RA & RB & 607 \\
0 & & & & \\
\end{array}

if \ RA = 0 \ then \ b \leftarrow 0 \\
else \ b \leftarrow (RA) \\
EA \leftarrow b + (RB) \\
FRT \leftarrow \text{MEM}(EA,8)

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into FRT.

For \text{lfdepx}, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRDS
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSR[GS] <E.HV>
- EPLCELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute \text{lfdepx} while MSR\_FP=0 will cause a Floating-Point Unavailable interrupt.

Corequisite Categories: Floating-Point

Special Registers Altered: None

---

Programming Note

This instruction behaves identically to a \text{lfdx} instruction except for using the EPLC register to provide the translation context.

---

Store Floating-Point Double by External Process ID Indexed X-form

\text{stfdepx} \ FRS,RA,RB

\begin{array}{ccc}
31 & FRS & RA & RB & 735 \\
0 & & & & \\
\end{array}

if \ RA = 0 \ then \ b \leftarrow 0 \\
else \ b \leftarrow (RA) \\
EA \leftarrow b + (RB) \\
\text{MEM}(EA,8) \leftarrow (FRS)

Let the effective address (EA) be the sum (RA|0)+(RB). (FRS) is stored into the doubleword in storage addressed by EA.

For \text{stfdepx}, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSCEPR is used in place of MSRPR
- EPSCEAS is used in place of MSRDS
- EPSCEPID is used in place of PID
- EPSCEGS is used in place of MSR[GS] <E.HV>
- EPSCELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute \text{stfdepx} while MSR\_FP=0 will cause a Floating-Point Unavailable interrupt.

Corequisite Categories: Floating-Point

Special Registers Altered: None

---

Programming Note

This instruction behaves identically to a \text{stfdx} instruction except for using the EPSC register to provide the translation context.
Vector Load Doubleword into Doubleword by External Process ID Indexed EVX-form

evlddepx RT,RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← MEM(EA,8)

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

For evlddepx, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC_EPR is used in place of MSRPR
- EPLC_EAS is used in place of MSR_DS
- EPLC_EPID is used in place of PID
- EPLC_EGS is used in place of MSR[GS] <E.HV>
- EPLC_ELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute evlddepx while MSR_SPV=0 will cause an SPE Unavailable interrupt.

Corequisite Categories:
Signal Processing Engine

Special Registers Altered:
None

Programming Note
This instruction behaves identically to a evlddx instruction except for using the EPLC register to provide the translation context.

Vector Store Doubleword into Doubleword by External Process ID Indexed EVX-form

evstddepx RS,RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,8) ← (RS)

Let the effective address (EA) be the sum (RA|0)+(RB). (RS) is stored into the doubleword in storage addressed by EA.

For evstddepx, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSCE_EPR is used in place of MSRPR
- EPSCE_EAS is used in place of MSR_DS
- EPSCE_EPID is used in place of PID
- EPSCE_EGS is used in place of MSR[GS] <E.HV>
- EPSCE_ELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute evstddepx while MSR_SPV=0 will cause an SPE Unavailable interrupt.

Corequisite Categories:
Signal Processing Engine

Special Registers Altered:
None

Programming Note
This instruction behaves identically to a evstddx instruction except for using the EPSC register to provide the translation context.
Load Vector by External Process ID
Indexed X-form

Ivepx  VRT,RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

For Ivepx, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:
- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRDS
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSR[GS] <E.HV>
- EPLCELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute Ivepx while MSR SPV=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

Programming Note
This instruction behaves identically to a lvx instruction except for using the EPLC register to provide the translation context.

Load Vector by External Process ID
Indexed LRU X-form

Ivepxl  VRT,RA,RB

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

Ivepxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

For Ivepxl, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:
- EPLCEPR is used in place of MSRPR
- EPLCEAS is used in place of MSRDS
- EPLCEPID is used in place of PID
- EPLCEGS is used in place of MSR[GS] <E.HV>
- EPLCELPID is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute Ivepxl while MSR SPV=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

Programming Note
See the Programming Notes for the lvxl instruction in Section 5.7.2 of Book I.

Programming Note
This instruction behaves identically to a lvxl instruction except for using the EPLC register to provide the translation context.
### Store Vector by External Process ID

#### Indexed X-form

<table>
<thead>
<tr>
<th>stvepx</th>
<th>VRS, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>VRS RA RB</td>
</tr>
<tr>
<td>807</td>
<td></td>
</tr>
</tbody>
</table>

- if RA = 0 then b ← 0
- else b ← (RA)
- EA ← b + (RB)
- MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

For **stvepx**, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSCEPR is used in place of MSRPR
- EPSCEAS is used in place of MSRDS
- EPSCEPID is used in place of PID
- EPSCEGS is used in place of MSR[GS] <E.HV>
- EPSCELPI is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute **stvepx** while MSR SPV=0 will cause a Vector Unavailable interrupt.

**Corequisite Categories:**
- Vector

**Special Registers Altered:**
- None

---

### Programming Note

This instruction behaves identically to a **stvx** instruction except for using the EPSC register to provide the translation context.

---

### Store Vector by External Process ID

#### Indexed LRU X-form

<table>
<thead>
<tr>
<th>stvepxl</th>
<th>VRS, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>VRS RA RB</td>
</tr>
<tr>
<td>775</td>
<td></td>
</tr>
</tbody>
</table>

- if RA = 0 then b ← 0
- else b ← (RA)
- EA ← b + (RB)
- MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

The **stvepxl** instruction provides a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

For **stvepxl**, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSCEPR is used in place of MSRPR
- EPSCEAS is used in place of MSRDS
- EPSCEPID is used in place of PID
- EPSCEGS is used in place of MSR[GS] <E.HV>
- EPSCELPI is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute **stvepxl** while MSR SPV=0 will cause a Vector Unavailable interrupt.

**Corequisite Categories:**
- Vector

**Special Registers Altered:**
- None

---

### Programming Note

See the Programming Notes for the **lvxl** instruction in Section 5.7.2 of Book I.

---

### Programming Note

This instruction behaves identically to a **stvxl** instruction except for using the EPSC register to provide the translation context.
Chapter 6. Storage Control

6.1 Overview

Instruction effective addresses are generated for sequential instruction fetches and for addresses that correspond to a change in program flow (branches, interrupts). Data effective addresses are generated by Load, Store, and Cache Management instructions. TLB Management instructions generate effective addresses to determine the presence of or to invalidate a specific TLB entry associated with that address. For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

Portions of the context of an effective address are appended to it to form the virtual address. The context is provided by various registers. The virtual address consists of the Logical Partition ID (LPID) <E.HV>, the Guest State <E.HV>, the address space identifier, the process identifier, and the effective address. The virtual address is translated to a real address by a matching “direct” entry in the Translation Lookaside Buffer (TLB) according to procedures described in Section 6.7.3. The Virtual Page Number (VPN) part of the virtual address is compared to the TLB contents to determine a match. The VPN consists of bits of the virtual address with the exception of the low-order effective address bits that correspond to the byte offset within the page. If the Embedded.Page Table category is supported, a virtual address can be translated by the Page Table pointed to by a matching “indirect” TLB entry as described in Section 6.7.4. As a result of a Page Table translation, a direct TLB entry is created, and this direct TLB entry can be used for subsequent translations. All virtual addresses are translated by the Page Table <E.PT> or the TLB, i.e. unlike the Server environment, there is no real mode. The real address that results from the translation is used to access main storage.

The Translation Lookaside Buffer is the hardware resource that also controls protection and storage control attributes. TLB permission bits control user and supervisor read, write and execute capability. If the Embedded.Hypervisor category is supported, the Virtualization Fault bit permits data accesses to pages to be trapped to the hypervisor, which allows the hypervisor to virtualize data accesses to specific pages, e.g. accesses to memory-mapped I/O. Storage control attributes described in Book II are supported by corresponding TLB bits, as are four optional implementation-dependent user-defined storage control attributes. The organization of the TLB (e.g. associativity, number of entries, number of arrays, etc.) is implementation-dependent. MMU configuration and TLB configuration information in various registers describes this implementation-dependent organization.

Software manages translation directly by installing TLB entries, and indirectly by setting up the page tables, which the TLB will cache. TLB Management instructions are used by software to read, write, search and invalidate TLB contents. MMU Assist Registers (MAS) are used to transfer data to and from the TLB arrays by TLB Management instructions. If the Embedded.Hypervisor category is not supported, TLB Management instructions are privileged instructions.

A different MMU Architecture Version (MAV) is used to indicate that different register layouts and functions are provided. The MMU Architecture Version Number is specified by the read-only MMUCFG register. The Embedded.Hypervisor.LRAT, Embedded TLB Write Conditional, and Embedded.Page Table categories are available only in MMU Architecture Version 2.0.

If the Embedded.Hypervisor category is supported and the Embedded.Hypervisor.LRAT category is not supported, most TLB Management instructions are hypervisor instructions. TLB entries contain real addresses, and, to maintain isolation between partitions, guest operating systems are not given access to real addresses. In this case most TLB Management instructions trap to the hypervisor. The hypervisor can emulate a TLB Management instruction by swapping a Real Page Number for corresponding Logical Page Number (LPN) in the MAS registers or vice versa so that the guest OS only sees LPNs.

However, if the Embedded.Hypervisor.LRAT category is supported, hardware can perform the translation of an LPN into a corresponding RPN. In this case, a TLB Write Entry (tlbwe) instruction can be executed in guest supervisor state. The LPN in the MAS register is translated into a corresponding RPN by a hardware lookup in a Logical to Real Address Translation (LRAT)
array, and the RPN is written to the TLB in place of the LPN when \texttt{tlbwe} is executed in guest supervisor state.

The Embedded.TLB Write Conditional (E.TWC) category provides a TLB write operation that is conditional on a TLB-reservation where the TLB-reservation is previously established by a \texttt{tlbsrx} instruction. The TLB-reservation is cleared by TLB invalidations and TLB writes involving the same virtual page. Thus, without acquiring a software lock, software can use the E.TWC category to write a TLB entry while ensuring that the entry is not a duplicate of an entry created simultaneously by another thread that shares the TLB and is not a stale value for a virtual page that was concurrently invalidated.

Figure 19 gives an overview of address translation if the Embedded.Page Table category is supported. The IND bit in a TLB entry indicates whether the entry is a "direct" entry or "indirect" entry. When a virtual address is translated, the TLB arrays are searched for a matching entry. If there is one and only one matching direct entry, that entry is used to translate the VA. If there is no matching direct TLB entry, but there is one and only one matching indirect entry, the indirect entry is used to access a Page Table Entry (PTE). If the PTE is a valid entry (V bit = 1), the PTE is used to translate the address and a "direct" entry is written to the TLB. If the Embedded.Page Table and Embedded.Hypervisor categories are both supported, the Embedded.Hypervisor.LRAT category is supported. In this case if the TGS bit of the indirect TLB entry is 1, the RPN from the PTE is treated as a Logical Page Number (LPN) and translated by the LRAT into an RPN. If the Embedded.Page Table is supported but the Embedded.Hypervisor category is not supported, supervisor software can create direct and indirect TLB entries and control the Page Table Entries. If both categories are supported, guest supervisor software can still create direct and indirect TLB entries and control the Page Table Entries if guest execution of \textit{TLB Management} instructions is enabled. However, depending on various factors such as the number of available LRAT entries, performance may be better if guest virtual addresses are translated by a Page Table that is managed by hypervisor software.

![Figure 19. Address translation with page table](image)

**Address Size Overview**

- Real address space size is $2^m$ bytes, $m\leq64$; see Note 1.
- In MMU Architecture Version 1.0, real page sizes are $4^p$ KB where $0\leq p \leq 15$ (i.e. 1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 64 MB, 256 MB, 1 GB, 4 GB, 16 GB, 64 GB, 256 GB, 1 TB); see Note 2. In MMU Architecture Version 2.0, real page sizes are $2^p$ KB where $0\leq p \leq 31$ (i.e. 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB, 8 GB, 16 GB, 32 GB, 64 GB, 128 GB, 256 GB, 512 GB, 1 TB, 2 TB); see Note 2. However, real pages sizes supported by a Page Table are limited to values of $p$ where $2\leq p \leq 15$.
- Effective address space size is $2^{64}$ bytes in 64-bit implementations and $2^{32}$ bytes in 32-bit implementations.
- The virtual address space size depends on the implementation.
6.2 Storage Exceptions

A storage exception results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Compare Debug Interrupt).

In certain cases a storage exception may result in the “restart” of (re-execution of at least part of) a Load or Store instruction. See Section 2.1 of Book II and Section 7.7 on page 1186 in this Book.

6.3 Instruction Fetch

For an instruction fetch, MSR\textsubscript{IS} is appended to the effective address as part of the virtual address. The Address Translation mechanism is described in Section 6.7.2, Section 6.7.3, and, if the Embedded.Page Table category is supported, Section 6.7.4.

6.3.1 Implicit Branch

Explicitly altering certain MSR bits (using \texttt{mtmsr}), or explicitly altering TLB entries, certain System Registers and possibly other implementation-dependent registers, may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an implicit branch. For example, an \texttt{mtmsr} instruction that changes the value of MSR\textsubscript{CM} may change the real address from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 12, “Synchronization Requirements for Context Alterations” on page 1235.

Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.
6.3.2 Address Wrapping Combined with Changing MSR Bit CM

If the current instruction is at effective address $2^{32}$.4 and is an \textit{mtsr} instruction that changes the contents of MSR$_{CM}$, the effective address of the next sequential instruction is undefined.

\begin{center}
\textbf{Programming Note}
\end{center}

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0, CSR0, or MCSRR0, as appropriate to the interrupt, are undefined if the Embedded.Hypervisor category is not supported or the interrupt is directed to the guest state. If the Embedded.Hypervisor category is supported and the interrupt is directed to the hypervisor state, the contents of GSRR0 are undefined.

6.4 Data Access

For a normal Load or Store instruction, MSR$_{DS}$ is appended to the effective address as part of the virtual address. The Address Translation mechanism is described in Section 6.7.2, Section 6.7.3, and, if the Embedded.Page Table category is supported, Section 6.7.4. The Embedded.External PID category must be supported. The effective address for an External Process ID Load or Store instruction data access is processed under control of the EPLC or EPSC, respectively. See Section 5.3.7.1 and Section 5.3.7.2.

6.5 Performing Operations Out-of-Order

An operation is said to be performed “in-order” if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed “out-of-order” if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as Branch, Trap, System Call, and Return From Interrupt instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, operations are performed out-of-order when resources are available that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, any results of the operation are abandoned (except as described below).

In the remainder of this section, including its subsections, “Load instruction” includes the Cache Management and other instructions that are stated in the instruction descriptions to be “treated as a Load”, and similarly for “Store instruction”.

A data access that is performed out-of-order may correspond to an arbitrary Load or Store instruction (e.g., a Load or Store instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

\begin{itemize}
  \item Stores
    Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).
  \item Accessing Guarded Storage
    The restrictions for this case are given in Section 6.8.1.1.
\end{itemize}

The only permitted side effects of performing an operation out-of-order are the following.

\begin{itemize}
  \item A Machine Check that could be caused by in-order execution may occur out-of-order except that, if category E.HV is supported and the Machine Check is the result of multiple TLB entries that translate the same VA, the Machine Check interrupt must occur in the context in which it was caused. Also, if category E.HV is supported, a Machine Check interrupt resulting from the following situations must be precise.
    \begin{itemize}
      \item Execution of an External Process ID instruction that has an operand that can be translated by multiple TLB entries.
      \item Execution of a \textit{tlbivax} instruction that isn't a TLB invalidate all and there are multiple entries in a single thread's TLB array(s) that match the complete VPN.
      \item Execution of a \textit{tlbli} instruction with T=3 and there are multiple entries in the TLB array(s) that match the complete VPN.
      \item Execution of a \textit{tlbsx} or \textit{tlbsrx} instruction and there are multiple matching TLB entries.
    \end{itemize}
  \item Non-Guarded storage locations that could be fetched into a cache by in-order fetching or execution of an arbitrary instruction may be fetched out-of-order into that cache.
\end{itemize}
6.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 6.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist. See Section 7.6.3 on page 1165.

6.7 Storage Control

This section describes the address translation facility, access control, and storage control attributes.

Demand-paged virtual memory is supported, as well as a variety of other management schemes that depend on precise control of effective-to-real address translation and flexible memory protection. Translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

The effective address space is divided into pages. The page represents the granularity of effective address translation, access control, and storage control attributes. In MMU Architecture Version 1.0, up to sixteen page sizes (1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB, 4GB, 16GB, 64GB, 256GB, 1TB) may be simultaneously supported. In MMU Architecture Version 2.0, up to 32 page sizes (1KB, 2 KB, 4 KB, 8 KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB, 8GB, 16GB, 32GB, 64GB, 128GB, 256GB, 512GB, 1TB, 2TB) may be simultaneously supported. In order for an effective to real translation to exist, a valid entry for the page containing the effective address must be in the Translation Lookaside Buffer (TLB). Addresses for which no TLB entry exists cause TLB Miss exceptions.

6.7.1 Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is the hardware resource that controls translation, protection, and storage control attributes. The organization of the TLB (e.g. associativity, number of entries, number of arrays, etc.) is implementation-dependent. Thus, the software for updating the TLB is also implementation-dependent. However, MMU configuration and TLB configuration information is provided such that software written to handle various TLB organizations could potentially run on multiple MMU implementations. A unified TLB organization (one to four TLB arrays, called TLB0, TLB1, TLB2 and TLB3, where each contains translations for both instructions and data) is assumed in the following description. For details on how to synchronize TLB updates with instruction execution see Section 6.11.4.3 and Chapter 12.

Maintenance of TLB entries is under software control, except that if the Embedded.Page Table category is supported, hardware will write TLB entries for translations performed via the Page Table. System software determines TLB entry replacement strategy and the format and use of any page state information. If a TLB provides Next Victim (NV) information, software can optionally use NV to choose a TLB entry to be replaced. See Section 6.11.4.7. Some implementations allow software to specify that a hardware generated hash and hardware replacement algorithm should be used to select the entry. See Section 6.11.4.7. The TLB entry contains all the information required to identify the page, to specify the translation, to specify access controls, and to specify the storage control attributes.

A TLB entry is written by copying information from MAS registers, using a tlbwe instruction (see page 1141). A TLB entry is read by copying information to MAS registers, using a tlbre instruction (see page 1139). Software can also search for specific TLB entries using the tlbsx instruction (see page 1136) and, if the Embedded.TLB Write Conditional category is supported, tlbxsx. (see page 1138).

Each TLB entry describes a page. Fields in the TLB entry fall into five categories:

- Page identification fields (information required to identify the page to the hardware translation mechanism).
- Address translation fields
- Access control fields
- Storage control attribute fields
- TLB management field

While the fields in the TLB entry are required, unless they are identified as part of a category that is not supported, no particular TLB entry format is formally specified. The tlbre and tlbwe instructions provide the ability to read or write individual entries. Below are shown the field definitions for the TLB entry. Some fields that are used only for indirect TLB entries can be overlaid with fields that are used only for direct TLB entries. Such overlap is implementation-dependent and an example is shown in Figure 20 on page 1081.
### Page Identification Fields for direct and indirect entries

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPN</td>
<td>Effective Page Number (up to 54 bits)</td>
</tr>
</tbody>
</table>

Bits 0:\(n\)–1 of the EPN field are compared to bits 0:\(n\)–1 of the effective address (EA) of the storage access (where \(n=64–\log_2(\text{page size in bytes})\) and \text{page size} is specified by the SIZE field of the TLB entry). See Table 2 and Table 3.

**Note:** Bits X:Y of the EPN field are implemented, where X=0 (64-bit implementation) or X=32 (32-bit implementation), and Y ≤ 53. \(Y = p - 1\) where \(p = 64–\log_2(\text{smallest page size in bytes})\) and \text{smallest page size} is the smallest page size supported by the implementation as specified by TLB array’s TLBnCFG or TLBnPS. The number of bits implemented for EPN is not required to be the same number of bits as are implemented for RPN. Implemented bits that represent offsets within a page are ignored for address comparisons performed for translation, invalidation, and searches and software should set these bits to zero. Unimplemented EPN bits are treated as if they contain 0s.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>Translation Address Space</td>
</tr>
</tbody>
</table>

This bit indicates the address space this TLB entry is associated with. For instruction storage accesses, MSRIS must match the value of TS in the TLB entry to provide the translation. Likewise, for data storage accesses, MSRDS must match the value of TS in the TLB entry. For the \texttt{tlbsrx. <E.TWC>} instruction, MAS1TS provides the address space specification that must match the value of TS. For the \texttt{tlbsx} instruction, MAS6SAS provides the address space specification that must match the value of TS. For the instructions \texttt{tiblix} with T=3 and \texttt{tibivax} with EA61=0, MAS6SAS provides the address space specification that is compared to the value of TS.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE</td>
<td>Page Size</td>
</tr>
</tbody>
</table>

For direct TLB entries, the SIZE field specifies the size of the virtual page associated with the TLB entry. For indirect TLB entries, the SIZE field specifies the maximum amount of virtual storage that can be mapped by the page table to which the indirect TLB entry points. The following applies in both cases:

- For MAV = 1.0, \(4^{\text{SIZEKB}}, 0 ≤ \text{SIZE} ≤ 15\). See Table 2. For TLB arrays that contain fixed-size TLB entries, this field is treated as reserved for \text{tlbwe} and \text{tlbre} instructions and is treated as a fixed value for translations. For variable page size TLB arrays, this field must be a value between TLBnCFG\_MINSIZE and TLBnCFG\_MAXSIZE.

- For MAV = 2.0, \(2^{\text{SIZEKB}}, 0 ≤ \text{SIZE} ≤ 31\). See Table 3. This field must be one of the page sizes specified by the corresponding TLBnPS register.

Implementations may support any one or more of the page sizes described above.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TID</td>
<td>Translation ID (implementation-dependent size)</td>
</tr>
</tbody>
</table>

Field used to identify a shared page (TID=0) or the owner’s process ID of a private page (TID≠0). See Section 6.7.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLPID</td>
<td>Translation Logical Partition ID &lt;E.HV&gt;</td>
</tr>
</tbody>
</table>

This field identifies a partition. The Translation Logical Partition ID is compared with LPIDR contents during translation. This allows for an efficient change of address space when a transition between partitions occurs. This number of bits in this field is an implementation-dependent number \(n\), where 1 ≤ \(n\) ≤ 12. See Section 6.7.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGS</td>
<td>Translation Guest State &lt;E.HV&gt;</td>
</tr>
</tbody>
</table>

This 1-bit field indicates whether this TLB entry is valid for the guest space or for the hypervisor space. The Translation Guest Space field is compared with the MSR\_GS bit during translation. This allows for an efficient change of address space when a transition from guest state to hypervisor state occurs. See Section 6.7.2.

- 0 Hypervisor space
- 1 Guest space
This bit indicates whether that this TLB entry is valid and may be used for translation. The Valid bit for a given entry can be set or cleared with a *tlbwe* instruction; alternatively, the Valid bit for an entry may be cleared by a *tlbilx* or *tlbivax* instruction or by a MMUCSR0 TLB invalidate all.

**IND Indirect <E.PT>**

This bit distinguishes between an indirect TLB entry that points to a Page Table (IND=1) and a direct TLB entry that can be used directly to translate a virtual address (IND=0). If a TLB array does not support this bit (TLBnCFGIND = 0), the implied IND value is 0. For the *tlbsx* instruction, MAS6SIND provides the direct/indirect specification that must match the value of IND. For the instructions *tlbilx* with T=3 and *tlbivax* with EA61=0, MAS6SIND provides the direct/indirect specification that is compared to the value of IND. See Section 6.7.4.

**Page Identification Field for indirect entry**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSIZE</td>
<td>Sub-Page Size (IND=1) &lt;E.PT&gt;</td>
</tr>
<tr>
<td></td>
<td>SPSIZE is a 5-bit field that specifies the minimum page size that can be specified by each Page Table Entry in the Page Table that is pointed to by the indirect TLB entry. This minimum page size is $2^{SPSIZE}$ KB and must be at least 4 KB. Thus SPSIZE must be at least 2. Valid values are specified by EPTCFG$_{SPS2 SPS1 SPS0}$. See Section 6.7.4.</td>
</tr>
</tbody>
</table>

**Translation Field**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPN</td>
<td><strong>Real Page Number</strong> (up to 54 bits)</td>
</tr>
<tr>
<td></td>
<td>For a direct TLB entry, bits $0:n-1$ of the RPN field are used to replace bits $0:n-1$ of the effective address to produce the real address for the storage access (where $n=$64−log$_2$(<em>page size in bytes</em>) and <em>page size</em> is specified by the SIZE field of the TLB entry). See Section 6.7.3 for a requirement on unused low-order RPN bits (i.e. bits $n:53$) being 0. For an indirect TLB entry, bits $0:m-1$ of the RPN field followed by 64- m 0s are the real address of the page table pointed to the indirect TLB entry, where $m=$ 61 − (SIZE − SPSIZE). RPN bits m:53 must be zero. See Section 6.7.4.</td>
</tr>
</tbody>
</table>

**Note:** Bits X:Y of the RPN field are implemented, where $X \geq 0$ and $Y \leq 53$. $X = 64 - MMUCFG_{RASIZE}$. $Y$ is the larger of the following applicable values:

- $p-1$ where $p=64-\log_2(\text{smallest page size in bytes})$ and *smallest page size* is the smallest page size supported by the implementation as specified by TLB array's TLBnCFG or TLBnPS.
- 52 if the Embedded.Page Table category is supported and a page table size of 2 KB is supported (EPTCFG$_{SPS2} = EPTCFG_{SPS1} = 8$ for some value of $n$).

The number of bits implemented for EPN is not required to be the same number of bits as are implemented for RPN. Unimplemented RPN bits are treated as if they contain 0s.

**Storage Control Bits** (see Section 6.8.3 on page 1097)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td><strong>Write-Through Required</strong></td>
</tr>
<tr>
<td></td>
<td>This bit indicates whether the page is Write-Through Required. See Section 1.6.1 of Book II.</td>
</tr>
<tr>
<td>0</td>
<td>This page is not Write-Through Required storage.</td>
</tr>
<tr>
<td>1</td>
<td>This page is Write-Through Required storage.</td>
</tr>
</tbody>
</table>

| I           | **Caching Inhibited**                                                       |
|             | This bit indicates whether the page is Caching Inhibited. See Section 1.6.2 of Book II. |
| 0           | This page is not Caching Inhibited storage.                                 |
| 1           | This page is Caching Inhibited storage.                                     |
M  Memory Coherence Required
   This bit indicates whether the page is Memory Coherence Required. See Section 1.6.3 of Book II.
   0  This page is not Memory Coherence Required storage.
   1  This page is Memory Coherence Required storage.

G  Guarded
   This bit indicates whether the page is Guarded. See Section 1.6.4 of Book II and Section 6.8.1.
   0  This page is not Guarded storage.
   1  This page is Guarded storage.

E  Endian Mode
   This bit indicates whether the page is accessed in Little-Endian or Big-Endian byte order. See Section 1.10.1 of Book I and Section 1.6.5 of Book II.
   0  The page is accessed in Big-Endian byte order.
   1  The page is accessed in Little-Endian byte order.

U0:U3  User-Definable Storage Control Attributes
       See Section 6.8.2.
       Specifies implementation-dependent and system-dependent storage control attributes for the page associated with the TLB entry. The existence of these bits is implementation-dependent.

VLE  Variable Length Encoding <E>VLE>
   This bit specifies whether a page which contains instructions is to be decoded as VLE instructions (see Chapter 1 of Book VLE). See Section 6.8.3 and Chapter 1 of Book VLE.
   0  Instructions fetched from the page are decoded and executed as non-VLE instructions.
   1  Instructions fetched from the page are decoded and executed as VLE instructions.

ACM  Alternate Coherency Mode
   This bit allows an implementation to employ more than a single coherency method. This allows participation in multiple coherency protocols. If the M attribute (Memory Coherence Required) is not set for a page (M=0), the page has no coherency associated with it and the ACM attribute is ignored. If the M attribute is set to 1 for a page (M=1), the ACM attribute is used to determine the coherency domain (or protocol) used. The coherency method used in Alternate Coherency Mode is implementation-dependent.

Access Control Fields for direct TLB entry

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UX  User State Execute Enable (IND=0)</td>
<td>See Section 6.7.6.1.</td>
</tr>
<tr>
<td>0  Instruction fetch and execution is not permitted from this page while MSRPR=1 and will cause an Execute Access Control exception type Instruction Storage interrupt.</td>
<td></td>
</tr>
<tr>
<td>1  Instruction fetch and execution is permitted from this page while MSRPR=1.</td>
<td></td>
</tr>
<tr>
<td>SX  Supervisor State Execute Enable (IND=0)</td>
<td>See Section 6.7.6.1.</td>
</tr>
<tr>
<td>0  Instruction fetch and execution is not permitted from this page while MSRPR=0 and will cause an Execute Access Control exception type Instruction Storage interrupt.</td>
<td></td>
</tr>
<tr>
<td>1  Instruction fetch and execution is permitted from this page while MSRPR=0.</td>
<td></td>
</tr>
<tr>
<td>UW  User State Write Enable (IND=0)</td>
<td>See Section 6.7.6.2.</td>
</tr>
<tr>
<td>0  Store operations, including dcba, dbcz, and dbczep are not permitted to this page when MSRPR=1 and will cause a Write Access Control exception. A Write Access Control exception will cause a Data Storage interrupt.</td>
<td></td>
</tr>
<tr>
<td>1  Store operations, including dcba, dbcz, and dbczep are permitted to this page when MSRPR=1.</td>
<td></td>
</tr>
<tr>
<td>SW  Supervisor State Write Enable (IND=0)</td>
<td>See Section 6.7.6.2.</td>
</tr>
<tr>
<td>0  Store operations, including dcba, dcbi, dcbz, and dcbzep are not permitted to this page when MSRPR=0. Store operations, including dcbi, dcbz, and dcbzep, will cause a Write Access Control exception. A Write Access Control exception will cause a Data Storage interrupt.</td>
<td></td>
</tr>
<tr>
<td>1  Store operations, including dcba, dcbi, dcbz, and dcbzep, are permitted to this page when MSRPR=0.</td>
<td></td>
</tr>
<tr>
<td>UR  User State Read Enable (IND=0)</td>
<td>See Section 6.7.6.3.</td>
</tr>
<tr>
<td>0  Load operations (including load-class Cache Management instructions) are not permitted from this page when MSRPR=1 and will cause a Read Access Control exception. A Read Access Control exception will cause a Data Storage interrupt.</td>
<td></td>
</tr>
<tr>
<td>1  Load operations (including load-class Cache Management instructions) are permitted from this page when MSRPR=1.</td>
<td></td>
</tr>
</tbody>
</table>
There are two separate address spaces supported. MSR\textsubscript{IS} and MSR\textsubscript{DS} are used to indicate the address space used for instruction and data accesses respectively. MSR\textsubscript{IS} and MSR\textsubscript{DS} can be set independently to access address space 0 or address space 1. TLB entries have a corresponding TS bit which is compared either to MSR\textsubscript{IS} or MSR\textsubscript{DS} for instruction and data accesses respectively to determine if the TLB entry is a match.

If the Embedded.Hypervisor category is supported, the above two address spaces exist for each logical parti-
tion and for both the guest and non-guest states within each logical partition. The Logical Partition ID Register identifies the partition and a field in the TLB entry (TLPID) specifies which partition that TLB entry is associated with. The Guest State (GS) bit in the Machine State Register identifies the guest state or non-guest state and a bit in the TLB entry (TGS) specifies which of these states that TLB entry is associated with.

Load, Store, Cache Management, and Branch instructions and next-sequential-instruction fetches produce a 64-bit effective address. A one-bit address space identifier and a process identifier are prepended to the effective address to form the virtual address. If the Embedded Hypervisor category is supported, this address is also prepended by a Logical Partition ID and Guest State bit. The Logical Partition ID is provided by the contents of LPIDR and the Guest State bit is provided by the MSRGS. For instruction fetches, the address space identifier is provided by MSRIS and the process identifier is provided by the contents of the Process ID Register. For data storage accesses, the address space identifier is provided by the MSRDS and the process identifier is provided by the contents of the Process ID Register.

Load, Store, Cache Management, and Branch instructions and next-sequential-instruction fetches produce a 64-bit effective address. A one-bit address space identifier and a process identifier are prepended to the effective address to form the virtual address. If the Embedded Hypervisor category is supported, this address is also prepended by a Logical Partition ID and Guest State bit. The Logical Partition ID is provided by the contents of LPIDR and the Guest State bit is provided by the MSRGS. For instruction fetches, the address space identifier is provided by MSRIS and the process identifier is provided by the contents of the Process ID Register. For data storage accesses, the address space identifier is provided by the MSRDS and the process identifier is provided by the contents of the Process ID Register.

Figure 21. Effective-to-Virtual-to-Real TLB Address Translation Flow

6.7.3 TLB Address Translation

A program references memory by using the effective address computed by the hardware when it executes a Load, Store, Cache Management, or Branch instruction, and when it fetches the next instruction. A virtual address is formed from the effective address as described in Section 6.7.2 and the virtual address is translated to a real address according to the procedures described in this section. The storage subsystem uses the real address for the access. All storage access effective addresses are compared to the Translation Address Space bit (TS), the Translation ID field (TID), and the value in the Effective Page Number field (EPN), respectively, of each TLB entry. If the Embedded Hypervisor category is supported, the Logical Partition ID and the Guest State bit are also compared to the Translation Logical Partition ID (TLPID) and Translation Guest State (TGS) of each TLB entry. Figure 22 illustrates the criteria for a virtual address to match a specific TLB entry for a direct TLB entry (IND = 0). See Section 6.7.4 for details on Page Table translation using an indirect TLB entry. The virtual address of a storage access matches a direct TLB entry if the first four following conditions are true, and, additionally, if the Embedded Page Table category is supported, the fifth condition is true, and, addi-
tionally, if the Embedded.Hypervisor category is supported, the last two conditions are true.

- The Valid bit of the TLB entry is 1.
- The value of the address specifier for the storage access (MSRIS for instruction fetches, MSRDS for data storage accesses) is equal to the value in the TS bit of the TLB entry.
- The value of the process identifier in the PID register is equal to the value in the TID field of the TLB entry or the value of the PID field of the TLB entry is equal to 0.
- The contents of bits 0:n-1 of the effective address of the storage access are equal to the value of bits 0:n-1 of the EPN field of the TLB entry (where n=64-\log_2(\text{page size in bytes}) and \text{page size} is specified by the value of the SIZE field of the TLB entry). See Table 2 and Table 3.
- One of the following conditions is true.
  - The TLB array supports the IND bit (TLBnCFGIND = 1) and the IND bit of the TLB entry is equal to 0.
  - The TLB array does not support the IND bit (TLBnCFGIND = 0).
  - Either the value of the logical partition identifier in LPIDR is equal to the value of the TLPID field of the TLB entry, or the value of the TLPID field of the TLB entry is equal to 0.
  - The value of the guest state bit (MSRGs) is equal to the value of the TGS bit of the TLB entry.

### Table 2: Page Size and Effective Address to TLB EPN Comparison for MAV = 1.0

<table>
<thead>
<tr>
<th>SIZE (4\text{SIZE}KB)</th>
<th>Page Size (2\text{SIZE}KB)</th>
<th>EA to EPN Comparison (bits 0:53–2:\text{SIZE})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>1KB</td>
<td>EPN0:53 = ? EA0:53</td>
</tr>
<tr>
<td>0b00001</td>
<td>2KB</td>
<td>EPN0:52 = ? EA0:52</td>
</tr>
<tr>
<td>0b00010</td>
<td>4KB</td>
<td>EPN0:51 = ? EA0:51</td>
</tr>
<tr>
<td>0b00011</td>
<td>8KB</td>
<td>EPN0:49 = ? EA0:49</td>
</tr>
<tr>
<td>0b00100</td>
<td>16KB</td>
<td>EPN0:48 = ? EA0:48</td>
</tr>
<tr>
<td>0b00101</td>
<td>32KB</td>
<td>EPN0:47 = ? EA0:47</td>
</tr>
<tr>
<td>0b00110</td>
<td>64KB</td>
<td>EPN0:46 = ? EA0:46</td>
</tr>
<tr>
<td>0b00111</td>
<td>128KB</td>
<td>EPN0:45 = ? EA0:45</td>
</tr>
<tr>
<td>0b01000</td>
<td>256KB</td>
<td>EPN0:44 = ? EA0:44</td>
</tr>
<tr>
<td>0b01001</td>
<td>512KB</td>
<td>EPN0:43 = ? EA0:43</td>
</tr>
<tr>
<td>0b01010</td>
<td>1MB</td>
<td>EPN0:42 = ? EA0:42</td>
</tr>
<tr>
<td>0b01011</td>
<td>2MB</td>
<td>EPN0:41 = ? EA0:41</td>
</tr>
<tr>
<td>0b01100</td>
<td>4MB</td>
<td>EPN0:40 = ? EA0:40</td>
</tr>
<tr>
<td>0b01101</td>
<td>8MB</td>
<td>EPN0:39 = ? EA0:39</td>
</tr>
<tr>
<td>0b01110</td>
<td>16MB</td>
<td>EPN0:38 = ? EA0:38</td>
</tr>
<tr>
<td>0b01111</td>
<td>32MB</td>
<td>EPN0:37 = ? EA0:37</td>
</tr>
<tr>
<td>0b10000</td>
<td>64MB</td>
<td>EPN0:36 = ? EA0:36</td>
</tr>
<tr>
<td>0b10001</td>
<td>128MB</td>
<td>EPN0:35 = ? EA0:35</td>
</tr>
<tr>
<td>0b10010</td>
<td>256MB</td>
<td>EPN0:34 = ? EA0:34</td>
</tr>
<tr>
<td>0b10011</td>
<td>512MB</td>
<td>EPN0:33 = ? EA0:33</td>
</tr>
<tr>
<td>0b10100</td>
<td>1GB</td>
<td>EPN0:32 = ? EA0:32</td>
</tr>
<tr>
<td>0b10101</td>
<td>2GB</td>
<td>EPN0:31 = ? EA0:31</td>
</tr>
<tr>
<td>0b10110</td>
<td>4GB</td>
<td>EPN0:30 = ? EA0:30</td>
</tr>
<tr>
<td>0b10111</td>
<td>8GB</td>
<td>EPN0:29 = ? EA0:29</td>
</tr>
<tr>
<td>0b11000</td>
<td>16GB</td>
<td>EPN0:28 = ? EA0:28</td>
</tr>
<tr>
<td>0b11001</td>
<td>32GB</td>
<td>EPN0:27 = ? EA0:27</td>
</tr>
<tr>
<td>0b11010</td>
<td>64GB</td>
<td>EPN0:26 = ? EA0:26</td>
</tr>
<tr>
<td>0b11011</td>
<td>128GB</td>
<td>EPN0:25 = ? EA0:25</td>
</tr>
<tr>
<td>0b11100</td>
<td>256GB</td>
<td>EPN0:24 = ? EA0:24</td>
</tr>
<tr>
<td>0b11101</td>
<td>512GB</td>
<td>EPN0:23 = ? EA0:23</td>
</tr>
<tr>
<td>0b11110</td>
<td>1TB</td>
<td>EPN0:22 = ? EA0:22</td>
</tr>
<tr>
<td>0b11111</td>
<td>2TB</td>
<td>EPN0:21 = ? EA0:21</td>
</tr>
</tbody>
</table>

### Table 3: Page Size and Effective Address to TLB EPN Comparison for MAV = 2.0

<table>
<thead>
<tr>
<th>SIZE (2\text{SIZE}KB)</th>
<th>EA to EPN Comparison (bits 0:53–2:\text{SIZE})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00000</td>
<td>EPN0:53 = ? EA0:53</td>
</tr>
<tr>
<td>0b00001</td>
<td>EPN0:52 = ? EA0:52</td>
</tr>
<tr>
<td>0b00010</td>
<td>EPN0:51 = ? EA0:51</td>
</tr>
<tr>
<td>0b00011</td>
<td>EPN0:49 = ? EA0:49</td>
</tr>
<tr>
<td>0b00100</td>
<td>EPN0:48 = ? EA0:48</td>
</tr>
<tr>
<td>0b00101</td>
<td>EPN0:47 = ? EA0:47</td>
</tr>
<tr>
<td>0b00110</td>
<td>EPN0:46 = ? EA0:46</td>
</tr>
<tr>
<td>0b00111</td>
<td>EPN0:45 = ? EA0:45</td>
</tr>
<tr>
<td>0b01000</td>
<td>EPN0:44 = ? EA0:44</td>
</tr>
<tr>
<td>0b01001</td>
<td>EPN0:43 = ? EA0:43</td>
</tr>
<tr>
<td>0b01010</td>
<td>EPN0:42 = ? EA0:42</td>
</tr>
<tr>
<td>0b01011</td>
<td>EPN0:41 = ? EA0:41</td>
</tr>
<tr>
<td>0b01100</td>
<td>EPN0:40 = ? EA0:40</td>
</tr>
<tr>
<td>0b01101</td>
<td>EPN0:39 = ? EA0:39</td>
</tr>
<tr>
<td>0b01110</td>
<td>EPN0:38 = ? EA0:38</td>
</tr>
<tr>
<td>0b01111</td>
<td>EPN0:37 = ? EA0:37</td>
</tr>
<tr>
<td>0b10000</td>
<td>EPN0:36 = ? EA0:36</td>
</tr>
<tr>
<td>0b10001</td>
<td>EPN0:35 = ? EA0:35</td>
</tr>
<tr>
<td>0b10010</td>
<td>EPN0:34 = ? EA0:34</td>
</tr>
<tr>
<td>0b10011</td>
<td>EPN0:33 = ? EA0:33</td>
</tr>
<tr>
<td>0b10100</td>
<td>EPN0:32 = ? EA0:32</td>
</tr>
<tr>
<td>0b10101</td>
<td>EPN0:31 = ? EA0:31</td>
</tr>
<tr>
<td>0b10110</td>
<td>EPN0:30 = ? EA0:30</td>
</tr>
<tr>
<td>0b10111</td>
<td>EPN0:29 = ? EA0:29</td>
</tr>
<tr>
<td>0b11000</td>
<td>EPN0:28 = ? EA0:28</td>
</tr>
<tr>
<td>0b11001</td>
<td>EPN0:27 = ? EA0:27</td>
</tr>
<tr>
<td>0b11010</td>
<td>EPN0:26 = ? EA0:26</td>
</tr>
<tr>
<td>0b11011</td>
<td>EPN0:25 = ? EA0:25</td>
</tr>
<tr>
<td>0b11100</td>
<td>EPN0:24 = ? EA0:24</td>
</tr>
<tr>
<td>0b11101</td>
<td>EPN0:23 = ? EA0:23</td>
</tr>
<tr>
<td>0b11110</td>
<td>EPN0:22 = ? EA0:22</td>
</tr>
<tr>
<td>0b11111</td>
<td>EPN0:21 = ? EA0:21</td>
</tr>
</tbody>
</table>

### Programming Note

An implementation need not support all page sizes.

If the virtual address of the storage access matches a TLB entry in accordance with the selection criteria specified in the preceding paragraph, the value of the Real Page Number field (RPN) of the matching TLB entry provides the real page number portion of the real address. Let \( n = 64 - \log_2(\text{page size in bytes}) \) where \text{page size} is specified by the SIZE field of the TLB entry. Bits \( n:63 \) of the effective address are appended to bits 0:n-1 of the 54-bit RPN field of the matching TLB entry to produce the 64-bit real address (i.e. \( \text{RA} = \text{RPN}_{n-1:n} \parallel \text{EA}_{n:63} \)) that is presented to main storage to perform the storage access. The page size is determined by the value of the SIZE field of the matching TLB entry. See Table 4 and Table 5. Depending on the page size, certain RPN bits of the matching TLB entry must be zero as shown in Table 4 and Table 5. Otherwise, it is implementation-dependent whether the
address translation is performed as if these RPN bits are 0 or as if the corresponding RA bits are undefined values, or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs. If the specified page size is not supported by the implementation's TLB array, it is implementation-dependent whether the address translation is performed as if the page size was a smaller size or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs.

Table 4: Real Address Generation for MAV = 1.0

<table>
<thead>
<tr>
<th>SIZE</th>
<th>Page Size (4^SIZE KB)</th>
<th>RPN Bits Required to be Equal to 0</th>
<th>Real Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0000</td>
<td>1 KB</td>
<td>none</td>
<td>RPN0:53</td>
</tr>
<tr>
<td>0b0001</td>
<td>4KB</td>
<td>RPN52:53=0</td>
<td>RPN0:51</td>
</tr>
<tr>
<td>0b0010</td>
<td>16KB</td>
<td>RPN50:53=0</td>
<td>RPN0:49</td>
</tr>
<tr>
<td>0b0011</td>
<td>64KB</td>
<td>RPN48:53=0</td>
<td>RPN0:47</td>
</tr>
<tr>
<td>0b0100</td>
<td>256KB</td>
<td>RPN46:53=0</td>
<td>RPN0:45</td>
</tr>
<tr>
<td>0b0101</td>
<td>1MB</td>
<td>RPN44:53=0</td>
<td>RPN0:43</td>
</tr>
<tr>
<td>0b0110</td>
<td>4MB</td>
<td>RPN42:53=0</td>
<td>RPN0:41</td>
</tr>
<tr>
<td>0b0111</td>
<td>16MB</td>
<td>RPN40:53=0</td>
<td>RPN0:39</td>
</tr>
<tr>
<td>0b1000</td>
<td>64MB</td>
<td>RPN38:53=0</td>
<td>RPN0:37</td>
</tr>
<tr>
<td>0b1001</td>
<td>256MB</td>
<td>RPN36:53=0</td>
<td>RPN0:35</td>
</tr>
<tr>
<td>0b1010</td>
<td>1GB</td>
<td>RPN34:53=0</td>
<td>RPN0:33</td>
</tr>
<tr>
<td>0b1011</td>
<td>4GB</td>
<td>RPN32:53=0</td>
<td>RPN0:31</td>
</tr>
<tr>
<td>0b1100</td>
<td>16GB</td>
<td>RPN30:53=0</td>
<td>RPN0:29</td>
</tr>
<tr>
<td>0b1101</td>
<td>64GB</td>
<td>RPN28:53=0</td>
<td>RPN0:27</td>
</tr>
<tr>
<td>0b1110</td>
<td>256GB</td>
<td>RPN26:53=0</td>
<td>RPN0:25</td>
</tr>
<tr>
<td>0b1111</td>
<td>1TB</td>
<td>RPN24:53=0</td>
<td>RPN0:23</td>
</tr>
</tbody>
</table>

A TLB Miss exception occurs if there is no valid matching direct entry in the TLB for the page specified by the virtual address (Instruction or Data TLB Error interrupt) and, if the Embedded.Page Table category is supported, there is no matching indirect entry (see Section 6.7.4). A TLB Miss exception for an instruction fetch will result in an Instruction TLB Miss exception type Instruction TLB Error interrupt. A TLB Miss exception for a data storage access will result in a Data TLB Miss exception type Data TLB Error interrupt. Although the possibility exists to place multiple direct and/or multiple indirect entries into the TLB that match a specific virtual address, assuming a set-associative or fully-associative organization, doing so is a programming error. Either one of the matching entries is used or a Machine Check exception occurs if there are multiple matching direct entries or multiple matching indirect entries for an instruction or data access.

The rest of the matching TLB entry provides the access control bits (UX, SX, UW, SW, UR, SR, VF), and stor-
age control attributes (ACM [implementation-dependent], VLE <VLE>, U0, U1, U2, U3, W, I, M, G, E) for the storage access. The access control bits and storage control attribute bits specify whether or not the access is allowed and how the access is to be performed. See Sections 6.7.6 and 6.11.4.

Figure 22. Address Translation: Virtual Address to direct TLB Entry Match Process

6.7.4 Page Table Address Translation [Category: Embedded.Page Table]

A hardware Page Table is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers. There can be many hardware Page Tables. Each Page Table is defined by an indirect TLB entry. An indirect TLB entry is an entry that has its IND bit equal to 1.

An indirect TLB entry matches the virtual address if all fields match per Section 6.7.4 except for the IND bit and the IND bit of the TLB entry is 1. If there is no matching direct TLB entry, but there is one and only one matching indirect entry, the indirect entry is used to access a Page Table Entry (PTE) if the VF bit of the indirect TLB entry is 0. If the VF bit of this indirect TLB entry is 1, a Virtualization Fault exception occurs. If the PTE is a valid entry (V bit = 1), the PTE is used to translate the address. The PTE includes the abbreviated RPN (ARPN), page size (PS), storage control (WIMGE), implementation-dependent bits, and storage access control bits (BAPR,C) that are used for the access. If the Embedded.Page Table and Embedded.Hypervisor categories are both supported, the Embedded.Hypervisor.LRAT category is supported. In this case, the RPN from the PTE is treated as a Logical Page Number (LPN) and the LPN is translated by the LRAT into an RPN. See Section 6.9. If there is more than one matching direct TLB entry or more than one matching indirect TLB entry, any one of the duplicate entries may be used or Machine Check exception may occur.

See Section 6.7.5 for the rules that software must follow when updating the Page Table.
Even when the Embedded Hypervisor category is supported, a Page Table can optionally be treated as a guest supervisor resource due to the LRAT.

If the Page Table is treated as a hypervisor resource, the Page Table must be placed in storage to which only the hypervisor has access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data. An LRAT identity mapping (LPN=RPN) can be used when the Page Tables are treated as hypervisor resources, especially if only one LRAT entry is provided. If the LRAT identity mapping converts LPNs into RPNs that extend beyond the memory given to the partition, the Page Table Entries still provide the hypervisor with a mechanism to limit a guest's accesses to memory assigned to the partition, assuming guest execution of TLB Management instructions is disabled.

If storage accesses are to scattered virtual pages, an Embedded Page Table could be sparsely used, and, in the worst case, there could be only one valid PTE in the Page Table. In this case it would be more efficient for software to directly load TLB entries rather than have both an indirect TLBE and a direct TLBE, which is loaded from the Page Table.
Figure 23. Page Table Translation

Logical Address translation performed if Category E.HV is supported and the TGS bit of the associated indirect TLB entry is 1.
Figure 23 depicts the Page Table translation for a matching indirect TLB entry (TLBE). The Page Table Entry that is used to translate the Effective Address is selected by a real address formed from some combination of RPN bits from the TLBE and some EA bits. The low-order m bits of the RPN field in the indirect TLB entry must be zeros, where m is \((\text{SIZE} - \text{SPSIZE}) - 7\).

SIZE minus SPSIZE must be greater than 7 (corresponding to a page table size of at least 2 KB; see below under “Page Table Size and Alignment”). The SIZE and SPSIZE fields of the TLBE determine which bits of the RPN and EA are used in the following manner.

1. EA\(_{23:51}\) are shifted right q bits, according to a decode of SPSIZE, to produce a 29-bit result S. The value of q is \((\text{SPSIZE} - 2)\). Bits shifted out of the rightmost bit position are lost.

2. A 21-bit EA mask is formed based on a decode of SIZE and SPSIZE. The EA mask is \((29 - (\text{SIZE} - \text{SPSIZE}))_0 || (\text{SIZE} - \text{SPSIZE})_{-81}\).

3. The EA mask from step 2 is ANDed with the high-order 21 bits of the shifted EA result \(S_{0:20}\) from step 1 to form a 21-bit result.

4. RPN\(_{22:52}\) from the indirect TLB entry is ORed with the 21-bit result from step 3 to form a 21-bit result R.

5. The real address of the PTE is formed as follows:
   \[
   \text{RA} = \text{TLBERPN}_{0:31} || R || S_{21:28} || 0b000
   \]

The doubleword addressed by the real address result from step 5 is the PTE used to translate the EA if the PTE is valid (Valid bit = 1). If the PTE is valid, PTE\(_{PS}\) must be greater than or equal to the SPSIZE of the associated indirect TLB entry and must be less than or equal to the SIZE of the associated indirect TLB entry. The real address (RA) result is formed by concatenating 0x000 with the ARPN\(_{0:31-p}\) from the PTE and with the low-order p bits of EA, where p is equal to \(\log_2(\text{page size specified by PTE}_{PS})\).

\[
\text{RA} = 0x000 || \text{ARPN}_{0:51-p} || \text{EA}_{64-p:63}
\]

However, if an implementation supports a real address with only r bits, \(r<52\), and either the Embedded.Hypervisor category is not supported or the TGS bit of the corresponding indirect TLB entry is 0, the high-order 52-r bits of PTE\(_{ARP}\) are ignored and treated as 0s. If the Embedded.Hypervisor category is supported, an implementation supports a logical address with only q bits, \(q<52\), and the TGS bit of the corresponding indirect TLB entry is 1, the high-order 52-q bits of PTE\(_{ARP}\) are ignored and treated as 0s.

If the Embedded.Hypervisor.LRAT category is supported and the TGS bit of the associated indirect TLB entry is 1, the RA formed from the PTE is treated as a logical real address and translated by the LRAT. If there is no matching entry in the LRAT, an LRAT Miss exception occurs. See Section 6.9. If an LRAT Error interrupt results from this exception, ESR\(_PT\) is set to 1.

If the Page Table Entry that is accessed is invalid (Valid bit = 0), a Page Table Fault exception occurs. An Execute, Read, or Write Access Control exception occurs if a valid PTE is found but the access is not allowed by the access control mechanism. These exceptions are types of Instruction Storage exception or Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access. See Section 7.6.4 and Section 7.6.5 for additional information about these and other interrupt types. For either of these interrupts caused by a Page Table Fault exception or Execute, Read, or Write Access Control exception due to PTE permissions, ESR\(_PT\) or GESR\(_PT\) is set to 1 (GESR\(_PT\) if the Embedded.Hypervisor LRAT category is supported and the interrupt is directed to the guest. Otherwise, ESR\(_PT\)).
Chapter 6. Storage Control

Version 2.07

1089

Storage Control Attributes for the Page Table

A Page Table must be located in storage that is Big-Endian, Memory Coherence Required, not Caching Inhibited and not Guarded. If the translation of a virtual address matches an indirect TLB entry that has its storage control attribute E bit equal to 1, M bit equal to 0, I bit equal to 1, or G bit = 1, it is implementation-dependent whether the translation is performed as if valid values were substituted for the invalid values or as if the entry doesn’t match, or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs. The Page Table is allowed to be located in storage that is Write Through Required or Not Write Through Required. However, the same W value must be used for a single thread’s indirect and direct TLB entries that map the same PTE. The Implementations may require specific values for ACM and U0:U3.

Ordering of Implicit Accesses to the Page Table

The definition of “performed” given in Books II and III-E applies also to the implicit accesses to the Page Table by the thread in performing address translation. Accesses for performing address translation are considered to be loads in this respect. These implicit accesses are ordered by the sync instruction with L=0 as described below.

The Synchronize instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer (sync with L=0 or L=1). This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers. The sync instruction with L=0 (sync) has the following additional properties.

- The sync instruction provides an ordering function for all stores to the Page Table caused by Store instructions preceding the sync instruction with respect to lookups of the Page Table that are performed, by the thread executing the sync instruction, after the sync instruction completes. Executing a sync instruction ensures that all such stores will be performed, with respect to the thread executing the sync instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table lookups, are performed with respect to that thread.

- In conjunction with the tlbivax and tlbsync instructions, the sync instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the tlbipvax instruction description on page 1141.

Page Table Entry

Each Page Table Entry (PTE) maps a VPN to an RPN. If the corresponding indirect TLB entry has an LPID <E.HV> or PID value of zero, multiple VPNs are mapped by a single PTE in a Page Table pointed to by

Programming Note

For instructions following a sync instruction, the memory barrier need not order implicit storage accesses for purposes of address translation.

Storage Control Attributes for the Page Table

A Page Table must be located in storage that is Big-Endian, Memory Coherence Required, not Caching Inhibited and not Guarded. If the translation of a virtual address matches an indirect TLB entry that has its storage control attribute E bit equal to 1, M bit equal to 0, I bit equal to 1, or G bit = 1, it is implementation-dependent whether the translation is performed as if valid values were substituted for the invalid values or as if the entry doesn’t match, or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs. The Page Table is allowed to be located in storage that is Write Through Required or Not Write Through Required. However, the same W value must be used for a single thread’s indirect and direct TLB entries that map the same PTE. The Implementations may require specific values for ACM and U0:U3.

Ordering of Implicit Accesses to the Page Table

The definition of “performed” given in Books II and III-E applies also to the implicit accesses to the Page Table by the thread in performing address translation. Accesses for performing address translation are considered to be loads in this respect. These implicit accesses are ordered by the sync instruction with L=0 as described below.

The Synchronize instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer (sync with L=0 or L=1). This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers. The sync instruction with L=0 (sync) has the following additional properties.

- The sync instruction provides an ordering function for all stores to the Page Table caused by Store instructions preceding the sync instruction with respect to lookups of the Page Table that are performed, by the thread executing the sync instruction, after the sync instruction completes. Executing a sync instruction ensures that all such stores will be performed, with respect to the thread executing the sync instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table lookups, are performed with respect to that thread.

- In conjunction with the tlbivax and tlbsync instructions, the sync instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the tlbipvax instruction description on page 1141.

Page Table Entry

Each Page Table Entry (PTE) maps a VPN to an RPN. If the corresponding indirect TLB entry has an LPID <E.HV> or PID value of zero, multiple VPNs are mapped by a single PTE in a Page Table pointed to by

Programming Note

For instructions following a sync instruction, the memory barrier need not order implicit storage accesses for purposes of address translation.
such an indirect TLB entry. Figure 24 shows the layout of a PTE.

Figure 24. Page Table Entry

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:39</td>
<td>ARPN</td>
<td>Abbreviated Real Page Number</td>
</tr>
<tr>
<td>40:44</td>
<td>WIMGE</td>
<td>Storage control attributes</td>
</tr>
<tr>
<td>45</td>
<td>R</td>
<td>Reference bit</td>
</tr>
<tr>
<td>46:49</td>
<td>impl-dep</td>
<td>Implementation-dependent</td>
</tr>
</tbody>
</table>

These bits can be used to support User-Definable Storage Control Attributes, ACM and VLE. These bits are used in any combination of the following or subsets of the following:

- 46:49 - User-Definable Storage Control Attributes (U0:U3)
- 48 - ACM
- 49 - VLE <VLE>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>SW0</td>
<td>Available for software use</td>
</tr>
<tr>
<td>51</td>
<td>C</td>
<td>Change bit</td>
</tr>
<tr>
<td>52:55</td>
<td>PS</td>
<td>Page Size (real)</td>
</tr>
<tr>
<td>56:61</td>
<td>BAP</td>
<td>Base Access Permission bits:</td>
</tr>
<tr>
<td>0</td>
<td>UX</td>
<td>Base UX</td>
</tr>
<tr>
<td>1</td>
<td>SX</td>
<td>Base SX</td>
</tr>
<tr>
<td>2</td>
<td>UW</td>
<td>Base UW</td>
</tr>
<tr>
<td>3</td>
<td>SW</td>
<td>Base SW</td>
</tr>
<tr>
<td>4</td>
<td>UR</td>
<td>Base UR</td>
</tr>
<tr>
<td>5</td>
<td>SR</td>
<td>Base SR</td>
</tr>
<tr>
<td>62</td>
<td>SW1</td>
<td>Available for software use</td>
</tr>
<tr>
<td>63</td>
<td>V</td>
<td>Entry valid (V=1) or invalid (V=0)</td>
</tr>
</tbody>
</table>

**Table 6: Storage Access Control Bits Derived from a Page Table Entry**

<table>
<thead>
<tr>
<th>Derived Storage Access Control</th>
<th>Page Table Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>UX</td>
<td>$BAP_0 &amp; R$</td>
</tr>
<tr>
<td>SX</td>
<td>$BAP_1 &amp; R$</td>
</tr>
<tr>
<td>UW</td>
<td>$BAP_2 &amp; R &amp; C$</td>
</tr>
<tr>
<td>SW</td>
<td>$BAP_3 &amp; R &amp; C$</td>
</tr>
<tr>
<td>UR</td>
<td>$BAP_4 &amp; R$</td>
</tr>
<tr>
<td>SR</td>
<td>$BAP_5 &amp; R$</td>
</tr>
</tbody>
</table>

---

**Programming Note**

Unlike many architectures, the R and C bits in a Page Table entry are not updated by hardware.

---

**Page Table Size and Alignment**

A Page Table's size is $8 \times 2^{(\text{SIZE} - \text{SPSIZE})}$, where SIZE is the page size specified by the SIZE field of the indirect TLB entry used to access the Page Table and SPSIZE is the sub-page size specified by the SPSIZE field of this indirect TLB entry. Page Table sizes smaller than 2 KB are not allowed and SPSIZE must be greater than or equal to 2. This implies that the Page Table size is $2^{k} \leq s \leq 4$ GB. The Page Table is aligned on a boundary that is a multiple of its size.

**TLB Update**

As a result of a Page Table translation, a corresponding direct TLB entry is created if no exception occurs, is optionally created if certain exceptions occur, and is not created if certain other exceptions occur.

If no exception occurs, a direct TLB entry is written to create an entry corresponding to the virtual address and the contents of the PTE that was used to translate the virtual address. In this case, hardware selects the TLB array and TLB entry to be written. Any TLB array
that meets all the following criteria can be selected by the hardware.

- The TLB array supports the page size specified by PTEPS.
- The TLB array can be loaded from the Page Table (TLBnCFGPT = 1).

If no TLB array can be selected based on these criteria, then a TLB Ineligible exception occurs. Hardware also selects the entry within the TLB array based on some implementation-dependent algorithm. However, a valid TLB entry with IPROT = 1 must not be overwritten. If all TLB entries that can be used for a specific virtual page have IPROT = 1, then a TLB Ineligible exception occurs. In the absence of a higher priority exception, an Instruction Storage or Data Storage interrupt occurs, depending on whether the Page Table translation was due to an instruction fetch or data access and ESRTLBI is set to 1.

It is implementation-dependent whether a TLB entry is written as a result of a Page Table translation if a Page Table Fault exception occurs, but, if written, the valid bit of the TLB entry is set to 0. It is implementation-dependent whether a TLB entry is written as a result of a Page Table translation if an Execute, Read, or Write Access Control exception occurs. If the Embedded Hypervisor category is supported, an interrupt caused by a Page Table Translation is directed to the hypervisor or guest as specified by the applicable EPCR bits (DSIGS and ISIGS), except that a DSI or ISI resulting from a TLBI is always directed to the hypervisor.

A TLB entry is not written as a result of a Page Table translation if an LRAT Miss exception occurs or a TLB Ineligible exception occurs.

If a TLB entry is written, the entry is written based on the values shown in Table 7.
If implementations write TLB entries for out-of-order Page Table translations, a mechanism for disabling such TLB updates must be provided by the implementation in order for software to preload a TLB array without the possibility of creating multiple direct entries for the same virtual address.

### Programming Note

As a hardware simplification the architecture allows a TLB entry to be written with the valid bit set to 0 if a Page Table Fault exception occurs. A replacement of a valid TLB entry by an invalid entry is typically not a significant performance impact since software often swaps in the virtual page and creates a valid PTE for the page.

---

### 6.7.5 Page Table Update Synchronization Requirements [Category: Embedded.Page Table]

This section describes rules that software must follow when updating the Page Table. Otherwise, TLB entries for outdated PTEs may remain valid. This section includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) that corresponds to a single line in the sequence is assumed to be done using a Store instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two Store Word instructions).

As described in Section 6.5, stores are not performed out-of-order. Moreover, address translations associated with instructions preceding the corresponding Store instructions are not performed again after the stores have been performed. (These address translations must have been performed before the store was determined to be required by the sequential execution model, because they might have caused an exception.) As a result, an update to a PTE need not be preceded by a context synchronizing operation.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the PTE are to be used for address translations associated with subsequent instructions.

As noted in the description of the Synchronize instruction in Section 4.4.3 of Book II, address translation...
associated with instructions which occur in program order subsequent to the Synchronize may actually be performed prior to the completion of the Synchronize. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

**Programming Note**
In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the rfi instruction that returns from the interrupt handler may provide the required context synchronization.

Page Table Entries must not be changed in a manner that causes an implicit branch.

### 6.7.5.1 Page Table Updates

When Page Tables are in use, TLBs are non-coherent caches of the Page Table. TLB entries must be invalidated explicitly with one of the methods described in Section 6.11.4.3.

Unsynchronized lookups in the Page Table continue even while it is being modified. Any thread, including a thread on which software is modifying the Page Table, may look in the Page Table at any time in an attempt to translate a virtual address. When modifying a PTE, software must ensure that the PTE's Valid bit is 0 if the PTE is inconsistent (e.g., if the BAP field is not correct for the current ARPN field).

The sequences of operations shown in the following subsections assume a multi-threaded processor environment. In a system consisting of only a single-threaded processor the tlbsync must be omitted, and the mbar that separates the tlbivax from the tlb-sync can be omitted. In a multi-threaded processor environment, when tlbilx is used instead of tlbivax in a Page Table update, the synchronization requirements are the same as when tlbivax is used in a system consisting of only a single-threaded processor.

---

**Programming Note**
For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the sync instruction at the end of the sequence should be followed by a Store instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the sync instruction ensures that if a Load instruction executed by another thread returns the chosen value from location X, the sequence's stores to the Page Table have been performed with respect to that other thread. The Load instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE.)

This Note assumes that the Page Table and location X are in storage that is Memory Coherence Required.

### 6.7.5.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The Valid bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```asm
PTEARPN, WIMGE, R, SN0, C, PS, BAP, SW1, V /* new values
sync /* order updates before next Page Table lookup and before next data access. */
```
On a 32-bit implementation, the following sequence can be used.

```
PTEARPN(0:31) ← new value  
mbar /* order 1st update before 2nd */  
PTEARPN[32:39],MULX,R,SW0,C,PS,BAP,SW1,V ← new values  
sync /* order updates before next  
   Page Table lookup and before  
   next data access. */
```

### 6.7.5.1.2 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTEV ← 0 /* (other fields don’t matter) */  
sync /* order update before tlbivax and  
   before next Page Table lookup */  
tlbivax(old_LPID,old_GS,old_PID,old_AS,old_VA,  
   old_ISIZE, old_IND) /* invalidate old translation*/  
mbar /* order tlbivax before tlbsync */  
tlbsync /* order tlbivax before sync */  
sync /* order tlbivax, tlbsync, and update  
   before next data access */
```

### 6.7.5.1.3 Modifying a Page Table Entry

#### General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the old PTE can be deleted and a new one added using the sequences described in the two preceding sections, in order to ensure that the translation instantiated by the old entry is no longer available, maintain a consistent state, modify the PTE, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

#### Modifying the SW0 and SW1 Fields

If the only change being made to a valid entry is to modify the SW0 or SW1 fields, the following sequence suffices because the SW0 and SW1 fields are not used by the thread.

```
loop:
   ldarx r1 /* load of PTE */  
   r1 ← new SW0,SW1 /* replace SW0,SW1 in r1*/  
   stdcx. PTE ← r1 /* store of PTE  
   if still reserved (new SW0 or SW1  
   values, other fields unchanged) */  
   bne- loop /* loop if lost reservation */
A lwarx/stdcx. pair (specifying the low-order word of the PTE) can be used instead of the ldarx/stdcx. pair shown above.
```

#### Modifying a Reference or Change Bit

If the only change being made to a valid entry is to modify the R bit, the C bit or both, the preceding sequence suffices if the precise instant that hardware Page Table translations use the new value doesn’t matter. Reference, Change, and Valid bits are in different bytes to facilitate the use of a Store instruction of a byte to modify a Reference or Change bit instead of a ldarx and stdcx. However, the correctness of doing so is a software issue beyond the scope of this architecture.

### 6.7.5.2 Invalidating an Indirect TLB Entry

The following sequence can be used to ensure that translations by a Page Table that is mapped via an indirect entry will no longer occur and that the storage used for the Page Table can then be re-used for other purposes.

```
for all valid PTEs mapped by the indirect TLB entry
   PTEV ← 0 /* (other fields don’t matter) */  
   sync /* order stores to PTEs */  
for all valid PTEs mapped by the indirect TLB entry
   tlbivax(old_LPID,old_GS,old_PID,old_AS,old_VA,  
   old_ISIZE, MAS6SIND = 0) /* invalidate old PTE translations*/  
   mbar /* order tlbivax before tlbsync */  
   tlbsync /* order tlbivax before sync */  
   sync /* order tlbivax, tlbsync, and update  
   before next data access to the storage  
   locations occupied by the Page Table  
   pointed to by the old indirect TLB */
```

### 6.7.6 Storage Access Control

After a matching TLB entry has been identified, the access control mechanism selectively grants execute access, read access, and write access separately for user mode versus supervisor mode. If the Embedded.Hypervisor category is supported, the access control mechanism selectively controls an access so that the access can be virtualized by the hypervisor if appropriate. Figure 25 illustrates the access control process and is described in detail in Sections 6.7.6.1 through 6.7.6.6.

An Execute, Read, or Write Access Control exception or Virtualization Fault exception occurs if the appropriate TLB entry is found but the access is not allowed by the access control mechanism (Instruction or Data Storage interrupt). See Section 7.6 for additional information about these and other interrupt types. In certain cases, Execute, Read, and Write Access Control exceptions and Virtualization Fault exceptions may result in the restart of (re-execution of at least part of) a Load or Store instruction.

Implementations may provide additional access control capabilities beyond those described here.
6.7.6.1 Execute Access

The UX and SX bits of the TLB entry control execute access to the page (see Table 8).

Instructions may be fetched and executed from a page in storage while in user state (MSR PR=1) if the UX access control bit for that page is equal to 1. If the UX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in user state.

Instructions may be fetched and executed from a page in storage while in supervisor state (MSR PR=0) if the SX access control bit for that page is equal to 1. If the SX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in supervisor state.

Instructions from no-execute storage may be in the instruction cache if they were fetched into that cache when their effective addresses were mapped to execute permitted storage. Software need not flush a page from the instruction cache before marking it no-execute.

Furthermore, if the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (i.e. UX=0 when MSR PR=1 or SX=0 when MSR PR=0), an Execute Access Control exception type Instruction Storage interrupt is taken.

6.7.6.2 Write Access

The UW and SW bits of the TLB entry control write access to the page (see Table 8).

Store operations (including Store-class Cache Management instructions) are permitted to a page in storage while in user state (MSR PR=1) if the UW access control bit for that page is equal to 1. If the UW access control bit is equal to 0, then execution of the Store instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

Store operations (including Store-class Cache Management instructions) are permitted to a page in storage while in supervisor state (MSR PR=0) if the SW access control bit for that page is equal to 1. If the SW access control bit is equal to 0, then execution of the Store instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

6.7.6.3 Read Access

The UR and SR bits of the TLB entry control read access to the page (see Table 8).

Load operations (including Load-class Cache Management instructions) are permitted from a page in storage while in user state (MSR PR=1) if the UR access control bit for that page is equal to 1. If the UR access control bit is equal to 0, then execution of the Load instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

Load operations (including Load-class Cache Management instructions) are permitted from a page in storage while in supervisor state (MSR PR=0) if the SR access control bit for that page is equal to 1. If the SR access...
control bit is equal to 0, then execution of the Load instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

6.7.6.4 Virtualized Access <E.HV>

The VF bit of the TLB entry prevents a Load or Store access to the page (see Table 8).

The translation of a Load or Store (including Cache Management instructions) operand address that uses a TLB entry with the Translation Virtualization Fault field equal to 1 causes a Virtualization Fault exception type Data Storage interrupt regardless of the settings of the permission bits and regardless of whether the TLB entry is a direct or indirect entry. The resulting Data Storage interrupt is directed to the hypervisor state.

6.7.6.5 Storage Access Control Applied to Cache Management Instructions

dcbi, dci, dcbz, and dcbzep instructions are treated as Stores since they can change data (or cause loss of data by invalidating dirty lines). As such, they can cause Write Access Control exception type Data Storage interrupts and Virtualization Fault exception type Data Storage interrupts. If an implementation first flushes a line before invalidating it during a dcbi, the dcbi is treated as a Load since the data is not modified.

dcba instructions are treated as Stores since they can change data. However, they do not cause Write Access Control exceptions. A dcba instruction will not cause a virtualization fault (TLBVF = 1).

dcbt, dcbtep, and icbt instructions are treated as Loads with respect to protection. However, they do not cause Read Access Control exceptions. A virtualization fault on these instructions will not result in a Data Storage interrupt.

dcbst and dcbstep instructions are treated as Stores with respect to protection. However, they do not cause Write Access Control exceptions. A virtualization fault on these instructions will not result in a Data Storage interrupt.

It is implementation-dependent whether dcbtstls instructions are treated as Loads or Stores with respect to protection. As such, they can cause either Load Access Control exception type Data Storage interrupts or Write Access Control exception type Data Storage interrupts and can also cause Virtualization Fault exception type Data Storage interrupts.

dcbf, dcbfep, dcbst, and dcbstep instructions are treated as Loads with respect to protection. Flushing or storing a line from the cache is not considered a Store since the store has already been done to update the cache and the dcbf, dcbfep, dcbst, or dcbstep instruction is only updating the copy in main storage. As a Load, they can cause Read Access Control exception type Data Storage interrupts and Virtualization Fault exception type Data Storage interrupts.

Table 8: Storage Access Control Applied to Cache Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Read Protection Violation</th>
<th>Write Protection Violation</th>
<th>Virtualization Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcbi</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>dcbf</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbfep</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbi</td>
<td>Yes3</td>
<td>Yes3</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbst</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbstep</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbt</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>dcbtep</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>dcbtls</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbst</td>
<td>No</td>
<td>Yes5</td>
<td>No</td>
</tr>
<tr>
<td>dcbstep</td>
<td>No</td>
<td>Yes5</td>
<td>No</td>
</tr>
<tr>
<td>dcbtstls</td>
<td>Yes4</td>
<td>Yes4</td>
<td>Yes4</td>
</tr>
<tr>
<td>dcbzd</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>dcbzep</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>dci</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>icbi</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>icbep</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>icblc</td>
<td>Yes2</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>icblq</td>
<td>Yes2</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>icbt</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>icbtls</td>
<td>Yes2</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ici</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

1. Category: Embedded.Hypervisor
2. icbtl and icbic require execute or read access.
3. dcbi may cause a Read or Write Access Control Exception based on whether the data is flushed prior to invalidation.
4. It is implementation-dependent whether dcbtstls is treated as a Load or a Store.
5. If an exception is detected, the instruction is treated as a no-op and no interrupt is taken.

6.7.6.6 Storage Access Control Applied to String Instructions

When the string length is zero, neither lswx nor stswx can cause Data Storage interrupts.
6.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

6.8.1 Guarded Storage

Storage is said to be “well-behaved” if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if the G bit is 1 in the TLB entry that translates the effective address.

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

Instruction fetching is not affected by the G bit.

The following rules apply to in-order execution of Load and Store Instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- Load or Store instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an asynchronous or imprecise interrupt is pending, the instruction completes before the interrupt occurs.

- Load or Store instruction that causes an Alignment exception, a Data TLB Error exception, or that causes a Data Storage exception.

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

6.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exception to this rule is the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

6.8.2 User-Definable

User-definable storage control attributes control user-definable and implementation-dependent behavior of the storage system. The existence of these bits is implementation-dependent. These bits are both implementation-dependent and system-dependent in their effect. These bits may be used in any combination and also in combination with the other storage control attribute bits.

6.8.3 Storage Control Bits

Storage control attributes are specified on a per-page basis. These attributes are specified in storage control bits in the TLB entries. The interpretation of their values is given in Figure 26.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Storage Control Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₁,₆</td>
<td>0 - not Write Through Required</td>
</tr>
<tr>
<td>I₆</td>
<td>0 - not Caching Inhibited</td>
</tr>
<tr>
<td>M₂</td>
<td>0 - not Memory Coherence Required</td>
</tr>
</tbody>
</table>

Programming Note

Instruction fetching from Guarded storage is permitted. If instruction fetches from Guarded storage must be prevented, software must set access control bits for such pages to no-execute (i.e. UX=0 and SX=0).
In Section 6.8.3.1 and 6.8.3.2, "access" includes accesses that are performed out-of-order.

### 6.8.3.1 Storage Control Bit Restrictions

All combinations of W, I, M, G, and E values are permitted except those for which both W and I are 1 and M || G ≠ \[0b10\].

The combination WIMG = \[0b1110\] is used to identify the Strong Access Ordering (SAO) storage attribute (see Section 1.7.1, “Storage Access Ordering”, in Book II). Setting WIMG=\[0b1110\] in a TLB entry causes accesses to the page to behave as if WIMG=\[0b0010\] with additional strong access ordering behavior. Only one SAO setting is provided because this attribute is not intended for general purpose programming, so a single combination of WIMG bits is supported.

If a given real page is accessed concurrently as SAO storage and as non-SAO storage, the result may be characteristic of the weakly consistent model.

**Programming Note**

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0. For implementations that support the SAO category, the operating system should provide a means by which application programs can request SAO storage, in order to avoid confusion with the preceding guideline (since SAO is encoded using \(W=0b11\)).

Accesses to the same storage location using two effective addresses for which the W bit differs meet the memory coherence requirements described in Section 1.6.3 of Book II if the accesses are performed by a single thread. If the accesses are performed by two or more threads, coherence is enforced by the hardware only if the W bit is the same for all the accesses.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

At any given time, data accesses to a given real page may use both Endian modes. When changing the Endian mode of a given real page for instruction fetching, care must be taken to prevent accesses while the change is made and to flush the instruction cache(s) after the change has been completed.

Setting the VLE attribute to 1 and setting the E attribute to 1 is considered a programming error and an attempt to fetch instructions from a page so marked produces an Instruction Storage Interrupt Byte Ordering Exception and sets ESRBO or GESRBO to 1 (GESRBO if the Embedded.Hypervisor category is supported and the interrupt is directed to the guest. Otherwise, ESRBO).

At any given time, the value of the VLE bit must be the same for all accesses to a given real page.

### References to Caching Inhibited storage

References to Caching Inhibited storage (or storage with \(I=1\)) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses \(I=1\). Conversely, references to storage that is not Caching Inhibited (or storage with \(I=0\)) apply to SAO storage or its WIMG encoding. References to Write Through Required storage (or storage with \(W=1\)) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the encoding using \(W=1\). Conversely, references to storage that is not Write Through Required (or storage with \(W=0\)) apply to SAO storage or its WIMG encoding.

**Figure 26. Storage control bits**

In Section 6.8.3.1 and 6.8.3.2, “access” includes accesses that are performed out-of-order.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Storage Control Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>0 - not Guarded 1 - Guarded</td>
</tr>
<tr>
<td>E(^3)</td>
<td>0 - Big-Endian 1 - Little-Endian</td>
</tr>
<tr>
<td>U0-U3(^4)</td>
<td>User-Definable</td>
</tr>
<tr>
<td>VLE(^5)</td>
<td>0 - non Variable Length Encoding (VLE). 1 - VLE</td>
</tr>
<tr>
<td>ACM(^6)</td>
<td>0 - not Alternate Coherency Mode 1 - Alternate Coherency Mode (if M=1)</td>
</tr>
</tbody>
</table>

1. Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.
2. Support of the 1 value is optional for implementations that do not support multiprocessing, implementations that do not support this storage control attribute assume the value of the bit to be 0, and setting M=1 in a TLB entry will have no effect.
3. [Category: Embedded.Little-Endian]
4. Support for these attributes is optional.
5. [Category: VLE]
6. [Category: SAO] The combination WIMG = \[0b1110\] has behavior unrelated to the meanings of the individual bits. See Section 6.8.3.1, “Storage Control Bit Restrictions” for additional information.
7. The coherency method used in Alternate Coherency Mode is implementation-dependent.

### Programming Note

In a system consisting of only a single-threaded processor that has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.
6.8.3.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using `dcbst`, `dcbstep`, `dcbf`, `dcbfep`, or `dcbi`.

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using `dcbf`, `dcbfep`, or `dcbi`, and `icbi` or `icbiep` before permitting any other accesses to the page.

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this to are system-dependent.

When changing the value of the ACM bit for a given real page are system-dependent.

When changing the value of the VLE bit for a given real page, software must set the VLE bit to the new value, then, if the page was not Caching Inhibited, invalidate copies of all locations in the page from instruction cache using `icbi` or `icbiep`, and then execute an `isync` instruction before permitting any other accesses to the page.
Programming Note

This Note suggests one example for managing reference and change recording.

When performing physical page management, it is useful to know whether a given physical page has been referenced or altered. Note that this may be more involved than knowing whether a given TLB entry has been used to reference or alter memory, since multiple TLB entries may translate to the same physical page. If it is necessary to replace the contents of some physical page with other contents, a page which has been referenced (accessed for any purpose) is more likely to be retained than a page which has never been referenced. If the contents of a given physical page are to be replaced, then the contents of that page must be written to the backing store before replacement, if anything in that page has been changed. Software must maintain records to control this process.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry to cast-out of the TLB, an entry which has been referenced is more likely to be retained in the TLB than an entry which has never been referenced.

Execute, Read and Write Access Control exceptions may be used to allow software to maintain reference information for a TLB entry and for its associated physical page. The entry is built, with its UX, SX, UR, SR, UW, and SW bits off, and the index and effective page number of the entry retained by software. The first attempt of application code to use the page will cause an Access Control exception (because the entry is marked “No Execute”, “No Read”, and “No Write”). The Instruction or Data Storage interrupt handler records the reference to the TLB entry and to the associated physical page in a software table, and then turns on the appropriate access control bit. An initial read from the page could be handled by only turning on the appropriate UR or SR access control bits, leaving the page “read-only”.

In a demand-paged environment, when the contents of a physical page are to be replaced, if any storage in that physical page has been altered, then the backing storage must be updated. The information that a physical page is dirty is typically recorded in a “Change” bit for that page.

Write Access Control exceptions may be used to allow software to maintain change information for a physical page. For the example just given for reference recording, the first write access to the page via the TLB entry will create a Write Access Control exception type Data Storage interrupt. The Data Storage interrupt handler records the change status to the physical page in a software table, and then turns on the appropriate UW and SW bits.
6.9 Logical to Real Address Translation [Category: Embedded.Hypervisor.LRAT]

In a partitioned environment, a guest operating system is not allowed to manipulate real page numbers. Instead the hypervisor virtualizes the real memory and the guest operating system manages the virtualized memory using logical page numbers (LPNs). In MMU Architecture Version 2.0, a Logical to Real Address Translation (LRAT) array facilitates this virtualization by providing a hardware translation from an LPN to an RPN without trapping to the hypervisor for every TLB update.

LRAT Entry

Below are shown the field definitions for an LRAT entry.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Valid This bit indicates that this LRAT entry is valid and may be used for translation of an LPN to an RPN. The Valid bit for a given entry can be set or cleared with a <code>tlbwe</code> instruction.</td>
</tr>
<tr>
<td>LPID</td>
<td>Logical Partition ID This optional field identifies a partition. The Logical Partition ID is compared with LPIDR contents during an LRAT translation. This field is required if category E.PT is supported or if threads that share an LRAT can be in different partitions. Whether the LPID field is supported is indicated by LRATCFGLPID. Note: The number of bits implemented for this field is required to be the same as the TLPID field in a TLB.</td>
</tr>
<tr>
<td>LPN</td>
<td>Logical Page Number (up to 54 bits) Bits 64-qn-1 of the LPN field are compared to bits 64-qn-1 of the Logical Page Number (LPN) for the <code>tlbwe</code> instruction or Page Table translation (where q = LRATCFG_LASIZE and n = 64 - log2(smallest logical page size in bytes) and logical page size is specified by the LSIZE field of the LRAT entry). See Section 6.7.2. Software must set unused low-order LPN bits to 0. Note: Bits X:Y of the LPN field are implemented, where X ≥ 0 and Y ≤ 53. The bits implemented for LPN are required to be the same as those implemented for TLBRPN. Unimplemented LPN bits are treated as if they contain 0s.</td>
</tr>
<tr>
<td>LSIZE</td>
<td>Logical Page Size The LSIZE field specifies the size of the logical page associated with the LRAT entry as 2^{LSIZE}, where 0 ≤ LSIZE ≤ 31. Implementations may support any one or more of these logical page sizes (see Section 6.10.3.6), and these logical page sizes need not be the same as the real page sizes that are implemented. However, the smallest logical page is no smaller than the smallest real page. The encodes for LSIZE are the same as the encodes for the TLB SIZE. See Section 6.7.2. This field must be one of the logical page sizes specified by the LRATPS register.</td>
</tr>
<tr>
<td>LRPN</td>
<td>LRAT Real Page Number (up to 54 bits) Bits 0:n-1 of the LRPN field are used to replace bits 0:n-1 of the LPN to produce the RPN that is written to TLBRPN by a <code>tlbwe</code> instruction or a Page Table translation (where n = 64 - log2(smallest logical page size in bytes) and logical page size is specified by the LSIZE field of the LRAT entry). Software must set unused low-order LRPN bits to 0. Note: Bits X:Y of the LRPN field are implemented, where X ≥ 0 and Y ≤ 53. X = 64 - MMUCFG_RASIZE, Y = p - 1 where p = 64 - log2(smallest logical page size in bytes) and smallest logical page size is the smallest page size supported by the implementation as specified by the LRATPS register. Unimplemented LRPN bits are treated as if they contain 0s.</td>
</tr>
</tbody>
</table>

An LRAT entry can be written by the hypervisor using the `tlbwe` instruction with MAS0_ATSEL equal to 1. The contents of the LRAT entry specified by MASDESEL and MAS2EPN are written from MAS registers. See the `tlbwe` instruction description in Section 6.11.4.9.

An LRAT entry can be read by the hypervisor using the `tlbre` instruction with MAS0_ATSEL equal to 1. The contents of the LRAT entry specified by MASDESEL and MAS2EPN are read and placed into the MAS registers. See the `tlbre` instruction description in Section 6.11.4.9.

Maintenance of LRAT entries is under hypervisor software control. Hypervisor software determines LRAT entry replacement strategy. There is no Next Victim support for the LRAT array.

The LRAT array is a hypervisor resource.

There is at most one LRAT array per thread.
Programming Note

Hypervisor software should not create an LRAT entry that maps any real memory regions for which a TLB entry should have VF equal to 1. Otherwise, a guest operating system could incorrectly create TLB entries, for this memory, with VF=0, assuming hypervisor software normally sets MAS8VF=0 before giving control to a guest operating system.

TLB Write

When the guest operating system manipulates the values of RPN fields of MAS registers, the values are treated as forming an LPN. When guest supervisor software attempts to execute `tlbwe` on an implementation that supports MMU Architecture Version 1, `tlbre`, and `tlbsx`, which operate on a TLB entry’s real page number (RPN) or when guest supervisor software attempts to execute a TLB Management instruction with guest execution of TLB Management instructions disabled (EPCR DGTMI=1), an Embedded Hypervisor Privilege exception occurs. Also, if TLBnCFG GTWE = 0 for a TLB array and the guest supervisor executes a `tlbwe` to the TLB array, an Embedded Hypervisor Privilege exception occurs. If a `tlbwe` caused the exception, the hypervisor can replace the LPN value in the MAS registers with the corresponding RPN, execute a `tlbwe`, and restore the LPN in the MAS registers before returning to the guest operating system. If a `tlbre` or `tlbsx` caused the exception, the hypervisor can execute the exception-causing instruction and replace the RPN value in the MAS registers with the corresponding LPN before returning to the guest operating system.

A Logical to Real Address Translation (LRAT) array provides a mechanism that allows a guest operating system to write the TLB without trapping to the hypervisor. When guest supervisor software executes `tlbwe` on an implementation that supports MMU Architecture Version 2, guest execution of TLB Management instructions is enabled (EPCR DGTMI=0), and TLBnCFG GTWE = 1 for the TLB array to be written, an LPN is formed. If MAS7 is implemented, LPN = MAS7RNNU || MAS3RPNL. Otherwise, LPN = 350 || MAS3RPNL. The LPN is translated into an RPN by the LRAT if a matching LRAT entry is found. A matching LRAT entry exists if the following conditions are all true for some LRAT entry:

The Valid bit of the LRAT entry is one.

Either the LPID field is not supported in the LRAT (LRATCFGLPID=0) or the value of LPIDR LPID is equal to the value of the LPID field of the LRAT entry.

Bits 64-q:n-1 of the LPN match the corresponding bits of the LPN field of the LRAT entry where n = 64 – \( \log_2(\text{logical page size in bytes}) \), \( \text{logical page size} \) is specified by the LSIZE field of the LRAT entry, and q is specified by LRATCFGLASIZE.

Either of the following is true:

- \( \text{MAS1}_{\text{IND}} = 0 \) and the value of \( \text{MAS1}_{\text{TSIZE}} \) is less than or equal to the value of the LSIZE field of the LRAT entry.
- \( \text{MAS1}_{\text{IND}} = 1 \) and the value of \( (3 + (\text{MAS1}_{\text{TSIZE}} - \text{MAS3SPSIZE})) \) is less than or equal to the value of the \( (10 + \text{LRAT entry}_{\text{LSIZE}}) \).

If a matching LRAT entry is found, the LRPN from that LRAT entry provides the upper bits of the RPN that is written to the TLB, and the LPN provides the low order RPN bits written to the TLB. Let \( n=64-\log_2(\text{logical page size in bytes}) \) where \( \text{logical page size} \) is specified by the LSIZE field of the LRAT entry. Bits \( n:53 \) of the LPN are appended to bits 0:n–1 of the LRPN field of the selected LRAT entry to produce the RPN (i.e., \( \text{RPN} = \text{LRPN}_{0:n-1} || \text{LPN}_{n:53} \)). The page size specified by the LSIZE of the LRAT entry used to translate the LPN must be one of the values supported by the implementation’s LRAT array. If the LRAT does not contain a matching entry for the LPN, an LRAT Miss exception occurs.

When the hypervisor executes a `tlbwe` instruction, no LRAT translation is performed and the RPN formed from MAS7RNNU and MAS3RPNL is written to the TLB.

Page Table

A Logical to Real Address Translation (LRAT) array provides a mechanism that allows guest Page Table management and translation without direct hypervisor involvement. When an instruction fetch address or a Load, Store, or Cache Management instruction operand address is translated by the Page Table, the Embedded.Hypervisor category is supported, and the TGS bit of the associated indirect TLB entry is 1, the RPN result of the Page Table translation is treated as an LPN that is translated into an RPN by the LRAT if a matching LRAT entry is found. A matching LRAT entry exists if the following conditions are all true for some LRAT entry:

The Valid bit of the LRAT entry is one.

Either the LPID field is not supported in the LRAT (LRATCFGLPID=0) or the value of LPIDR LPID is equal to the value of the LPID field of the LRAT entry.

Bits 64-q:n-1 of the LPN match the corresponding bits of the LPN field of the LRAT entry where \( n = 64 – \log_2(\text{logical page size in bytes}) \), \( \text{logical page size} \) is specified by the LSIZE field of the LRAT entry, and q is specified by LRATCFGLASIZE.

The value of PTEPS is less than or equal to the value of the LSIZE field of the LRAT entry.

If a matching LRAT entry is found, the LRPN from that LRAT entry provides the upper bits of the RPN of the translation result and the LPN provides the low order RPN bits of the translation result. Let \( n=64-\log_2(\text{logical page size in bytes}) \) where \( \text{logical page size} \) is specified by the LSIZE field of the LRAT entry. Bits
Chapter 6. Storage Control

6.10 Storage Control Registers

In addition to the registers described below, the Machine State Register provides the IS and DS bits, that specify which of the two address spaces the respective instruction or data storage accesses are directed towards. MSRPR bit is also used by the storage access control mechanism. If the Embedded.Hypervisor category is supported, the MSRGS bit is used to identify guest state. The guest supervisor state exists when MSRPR = 0 and MSRGS = 1. MSRGS is used to form the virtual address. Also, see Section 5.3.7 for the registers in the Embedded.External PID category.

6.10.1 Process ID Register

The Process ID Registers are 32-bit registers as shown in Figure 27. Process ID Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The number of bits implemented in a PID register is indicated by the value of the MMUCFG_PIDSIZE. The Process ID Register provides a value that is used to construct a virtual address for accessing storage.

The Process ID Register is a privileged register. This register can be read using mfspr and can be written using mtspr. An implementation may opt to implement only the least-significant \( n \) bits of the Process ID Register, where \( 1 \leq n \leq 14 \), and \( n \) must be the same as the number of implemented bits in the TID field of the TLB entry. The most-significant \( 32-n \) bits of the Process ID Register are treated as reserved.

Figure 27. Processor ID Register (PID)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>50:63</td>
<td>Processor ID (PID)</td>
</tr>
</tbody>
</table>

Identifies a unique process (except for the value of 0) and is used to construct a virtual address for storage accesses.

All other fields are reserved.

6.10.2 MMU Assist Registers

The MMU Assist Registers (MAS) are used to transfer data to and from the TLB arrays. If the Embedded.Hypervisor.LRAT category is supported, MAS registers are also used to transfer data to and from the LRAT array. MAS registers can be read and written by software using mfspr and mtspr instructions. Execution of a tlbre instruction with MAS0_ATSEL=0 causes the TLB entry specified by MAS0_TLBSEL, MAS0_ESEL, and MAS2_EPN to be copied to the MAS registers if TLBnCFGHES = 0 for the TLB array specified by MAS0_TLBSEL whereas the TLB entry is specified by MAS0_ESEL, MAS0_TLBSEL, and a hardware generated hash based on MAS2_EPN, MAS1_TID, and MAS1_TSIZE if TLBnCFGHES=1. Execution of a tlbwe instruction with MAS0_ATSEL=0 (or in guest supervisor state) causes the TLB entry specified by MAS0_TLBSEL, MAS0_ESEL, and MAS2_EPN to be written with contents of the MAS registers if TLBnCFGHES = 0 for the TLB array specified by MAS0_TLBSEL. If TLBnCFGHES = 1 for a tlbwe, the TLB entry is selected by MAS0_TLBSEL, a hardware generated hash based on MAS2_EPN, MAS1_TID, and MAS1_TSIZE, and either a hardware replacement algorithm if MAS0_HES=1 or MAS0_ESEL if MAS0_HES=0.

MAS registers may also be updated by hardware as the result of any of the following.

- A tlbx instruction
- The occurrence of an Instruction or Data TLB Error interrupt if any of the following is true.
  - The Embedded.Hypervisor category is not supported.
  - MAS Register updates are enabled for interrupts directed to the hypervisor (EPCR_MUH = 0).
  - The interrupt is directed to the guest state (EPCR_TLBGS = 1 for Instruction TLB Error interrupt and EPCR_DTLBGS = 1 for Data TLB Error interrupt).

All MAS registers are privileged, except MAS5 and MAS8, which are hypervisor privileged and are only provided if Category: Embedded.Hypervisor is supported. All MAS registers with the exception of MAS7 and, if Embedded.Hypervisor category is not supported, MAS5 and MAS8, must be implemented. MAS7 is not required to be implemented if the hardware supports 32 bits or less of real address.

The necessary bits of any multi-bit field in a MAS register must be implemented such that only the resources supported are represented. Any non-implemented bits in a field should have no effect when writing and should always read as zero. For example, if only 2 TLB arrays...
are implemented, then only the lower-order bit of the MAS0TLBSEL field is implemented.

**Programming Note**

Operating system developers should be wary of new implementations silently ignoring unimplemented MAS bits on MAS register writes. This is a common error during initial bring-up of a new processor.

### 6.10.3 MMU Configuration and Control Registers

#### 6.10.3.1 MMU Configuration Register (MMUCFG)

The read-only MMUCFG register provides information about the MMU and its arrays. MMUCFG is a privileged register except that if the Embedded.Hypervisor category is supported, MMUCFG is a hypervisor resource. The layout of the MMUCFG register is shown in Figure 28 for MAV=1.0 and in Figure 29 for MAV=2.0.

**Figure 28. MMU Configuration Register [MAV=1.0]**

```
+----------+----------+----------+----------+----------+----------+
| 32 36 40 47 53 58 60 62 63 |
| LPIDSIZE | RASIZE   | PIDSIZE  | NTLBS    | MAVN     |
+----------+----------+----------+----------+----------+
```

**Figure 29. MMU Configuration Register [MAV=2.0]**

```
+----------+----------+----------+----------+----------+----------+
| 32 36 40 47 48 49 53 58 60 62 63 |
| LPIDSIZE | RASIZE   | LRAT     | TWC      | PIDSIZE  | NTLBS    | MAVN     |
+----------+----------+----------+----------+----------+----------+----------+
```

The MMUCFG fields are described below.

**Bit Description**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>36:39</td>
<td><strong>LPID Register Size</strong> (LPIDSIZE) The value of LPIDSIZE is the number of bits in LPIDR that are implemented. Only the least significant LPIDSIZE bits in LPIDR are implemented. The Embedded.Hypervisor category is supported if and only if LPIDSIZE &gt; 0.</td>
</tr>
<tr>
<td>40:46</td>
<td><strong>Real Address Size</strong> (RASIZE) Number of bits in a real address supported by the implementation.</td>
</tr>
<tr>
<td>47</td>
<td><strong>LRAT Translation Supported</strong> (LRAT) [Category: Embedded.Hypervisor.LRAT] Indicates LRAT translation is supported. 0 LRAT translation is not supported. A tlbw executed in guest supervisor state results in an Embedded Hypervisor Privilege exception.</td>
</tr>
<tr>
<td>53</td>
<td><strong>PID Register Size</strong> (PIDSIZE) The value of PIDSIZE is one less than the number of bits implemented for each of the PID registers implemented. Only the least significant PIDSIZE+1 bits in the PID registers are implemented. The maximum number of PID register bits that may be implemented is 14.</td>
</tr>
<tr>
<td>60:61</td>
<td><strong>Number of TLBs</strong> (NTLBS) The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented. NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of MAS0TLBSEL.</td>
</tr>
<tr>
<td>62:63</td>
<td><strong>MMU Architecture Version Number</strong> (MAVN) Indicates the version number of the architecture of the MMU implemented.</td>
</tr>
</tbody>
</table>

All other fields are reserved.

#### 6.10.3.2 TLB Configuration Registers (TLBnCFG)

Each TLBnCFG read-only register provides configuration information about each corresponding TLB array that is implemented. There is one TLBnCFG register implemented for each TLB array that is implemented. TLBnCFG corresponds to TLBn for 0 ≤ n ≤ MMUCFGNTLBS. TLBnCFG registers are privileged registers except that if the Embedded.Hypervisor category is supported, TLBnCFG registers are hypervisor resources. The layout of the TLBnCFG registers is...
shown in Figure 30 for MAV=1.0 and in Figure 31 for MAV=2.0.

Table 6-1: TLB Configuration Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:39</td>
<td>Associativity (ASSOC)</td>
</tr>
<tr>
<td></td>
<td>Total number of entries in a TLB array which can be used for translating</td>
</tr>
<tr>
<td></td>
<td>addresses with a given EPN. This number is referred to as the associativity</td>
</tr>
<tr>
<td></td>
<td>level of the TLB array. Some values of assoc have special meanings when</td>
</tr>
<tr>
<td></td>
<td>used in combination with specific values of NENTRY, as follows. However,</td>
</tr>
<tr>
<td></td>
<td>if TLBnCFG-GHES=1, the associativity level of the TLB array is implementation</td>
</tr>
<tr>
<td></td>
<td>dependent.</td>
</tr>
<tr>
<td>40:43</td>
<td>Minimum Page Size (MINSIZE) [MAV=1.0]</td>
</tr>
<tr>
<td></td>
<td>This field defines the minimum page size of the TLB array. Page size encoding</td>
</tr>
<tr>
<td></td>
<td>is defined in Section 6.7.2.</td>
</tr>
<tr>
<td>44:47</td>
<td>Maximum Page Size (MAXSIZE) [MAV=1.0]</td>
</tr>
<tr>
<td></td>
<td>This field defines the maximum page size of the TLB array. Page size encoding</td>
</tr>
<tr>
<td></td>
<td>is defined in Section 6.7.2.</td>
</tr>
<tr>
<td>45</td>
<td>Page Table (PT) [MAV=2.0 and Category: E.PT]</td>
</tr>
</tbody>
</table>

Figure 30. TLB Configuration Register [MAV=1.0]

Figure 31. TLB Configuration Register [MAV=2.0]

The TLBnCFG fields are described below.

Bit   Description
-------------------------------------------------------------------------
32:39 Associativity (ASSOC)
       Total number of entries in a TLB array which can be used for       |
       translating addresses with a given EPN. This number is referred   |
       to as the associativity level of the TLB array. Some values of    |
       assoc have special meanings when used in combination with specific |
       values of NENTRY, as follows. However, if TLBnCFG-GHES=1, the    |
       associativity level of the TLB array is implementation dependent. |
40:43 Minimum Page Size (MINSIZE) [MAV=1.0]
       This field defines the minimum page size of the TLB array. Page   |
       size encoding is defined in Section 6.7.2.                         |
44:47 Maximum Page Size (MAXSIZE) [MAV=1.0]
       This field defines the maximum page size of the TLB array. Page    |
       size encoding is defined in Section 6.7.2.                         |
45    Page Table (PT) [MAV=2.0 and Category: E.PT]                     |
       Indicates that the TLB array can be loaded from the hardware Page   |
       Table.
0    TLB array cannot be loaded from the Page Table.
1    TLB array can be loaded from the Page Table.

Indirect (IND) [MAV=2.0 and Category: E.PT] indicates that an indirect TLB entry can be created in the TLB array and that there is a corresponding EPTCFG register that defines the SIZE and Sub-Page Size values that are supported.
0 The TLB array treats the IND bit as reserved.
1 The TLB array supports indirect entries.

Guest TLB Write Enabled (GTWE) [MAV=2.0 and Category: Embedded.Hypervisor.LRAT] indicates that a guest supervisor can write the TLB array because LRAT translation is supported for the TLB array.
0 A guest supervisor cannot write the TLB array. A tlbwe executed in guest supervisor state results in an Embedded Hypervisor Privilege exception.
1 A guest supervisor can write the TLB array if guest execution of TLB Management instructions is enabled (EPCRDGTMI=0).

Invalidate Protection (IPROT) Invalidate protect capability of the TLB array.
0 Indicates invalidate protection capability not supported.
1 Indicates invalidate protection capability supported.

Page Size Availability (AVAIL) [MAV=1.0]
This defines the page size availability of the TLB array. If the Embedded.Page Table category is supported, this also defines the virtual address space size availability of TLB array. Otherwise, this field is reserved.
0 Fixed selectable page size from MINSIZE to MAXSIZE (all TLB entries are the same size).
1 Variable page size from MINSIZE to MAXSIZE (each TLB entry can be sized separately).

Hardware Entry Select (HES) [MAV=2.0]
Indicates whether the TLB array supports MAS0HES and the associated method for hardware selecting a TLB entry based on MAS1TID TSIZE and MAS2EPN for a tlbwe instruction.
0  MAS0HES is not supported.
1  MAS0HES is supported for tlbwe. See Section 6.10.3.8. For tlbre, MAS0ESSEL selects among the TLB entries that can be used for translating addresses with a given MAS1TID TSIZE and MAS2EPN. The set of TLB entries is determined by a hardware generated hash based on MAS1TID TSIZE and MAS2EPN. The hash is the same for tlbwe and tlbre for a given TLB array but could be different for each TLB array.

52:63  **Number of Entries (NENTRY)**

Number of entries in the TLB array.

All other fields are reserved.

### 6.10.3.3 TLB Page Size Registers (TLBnPS) [MAV=2.0]

Each TLBnPS read-only register provides page size information about each corresponding TLB array that is implemented in MMU Architecture Version 2.0. Each Page Size bit (PS0-PS31) that is a one indicates that a specific page size is supported by the array. Multiple 1 bits indicate that multiple page sizes are supported concurrently. TLBnPS registers are privileged registers except that if the Embedded.Hypervisor category is supported, TLBnPS registers are hypervisor resources. The layout of the TLBnPS registers is shown in Figure 32.

**Figure 32. TLB n Page Size Register**
The TLBnPS fields are described below.

#### Bit Description

32:63  **Page Size 31 - Page Size 0 (PS31-PS0)**

PSm indicates whether a direct TLB entry page size of 2^m KB is supported by the TLB array. PSm corresponds to bit TLBnPS_{63-m} for m = 0 to 31.

0  Direct TLB entry page size of 2^m KB is not supported.
1  Direct TLB entry page size of 2^m KB is supported.

Table 9 shows the relationship between the Page Size (PSm) bits in TLBnPS and page size. The existence and type of mechanism for configuring the use of a subset of supported page sizes is implementation-dependent.

<table>
<thead>
<tr>
<th>TLBnPS or LRATPS bit</th>
<th>PSm</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>PS31</td>
<td>2TB</td>
</tr>
<tr>
<td>33</td>
<td>PS30</td>
<td>1TB</td>
</tr>
<tr>
<td>34</td>
<td>PS29</td>
<td>512GB</td>
</tr>
<tr>
<td>35</td>
<td>PS28</td>
<td>256GB</td>
</tr>
<tr>
<td>36</td>
<td>PS27</td>
<td>128GB</td>
</tr>
<tr>
<td>37</td>
<td>PS26</td>
<td>64GB</td>
</tr>
<tr>
<td>38</td>
<td>PS25</td>
<td>32GB</td>
</tr>
<tr>
<td>39</td>
<td>PS24</td>
<td>16GB</td>
</tr>
<tr>
<td>40</td>
<td>PS23</td>
<td>8GB</td>
</tr>
<tr>
<td>41</td>
<td>PS22</td>
<td>4GB</td>
</tr>
<tr>
<td>42</td>
<td>PS21</td>
<td>2GB</td>
</tr>
<tr>
<td>43</td>
<td>PS20</td>
<td>1GB</td>
</tr>
<tr>
<td>44</td>
<td>PS19</td>
<td>512MB</td>
</tr>
<tr>
<td>45</td>
<td>PS18</td>
<td>256MB</td>
</tr>
<tr>
<td>46</td>
<td>PS17</td>
<td>128MB</td>
</tr>
<tr>
<td>47</td>
<td>PS16</td>
<td>64MB</td>
</tr>
<tr>
<td>48</td>
<td>PS15</td>
<td>32MB</td>
</tr>
<tr>
<td>49</td>
<td>PS14</td>
<td>16MB</td>
</tr>
<tr>
<td>50</td>
<td>PS13</td>
<td>8MB</td>
</tr>
<tr>
<td>51</td>
<td>PS12</td>
<td>4MB</td>
</tr>
<tr>
<td>52</td>
<td>PS11</td>
<td>2MB</td>
</tr>
<tr>
<td>53</td>
<td>PS10</td>
<td>1MB</td>
</tr>
<tr>
<td>54</td>
<td>PS09</td>
<td>512KB</td>
</tr>
<tr>
<td>55</td>
<td>PS08</td>
<td>256KB</td>
</tr>
<tr>
<td>56</td>
<td>PS07</td>
<td>128KB</td>
</tr>
<tr>
<td>57</td>
<td>PS06</td>
<td>64KB</td>
</tr>
<tr>
<td>58</td>
<td>PS05</td>
<td>32KB</td>
</tr>
<tr>
<td>59</td>
<td>PS04</td>
<td>16KB</td>
</tr>
<tr>
<td>60</td>
<td>PS03</td>
<td>8KB</td>
</tr>
<tr>
<td>61</td>
<td>PS02</td>
<td>4KB</td>
</tr>
<tr>
<td>62</td>
<td>PS01</td>
<td>2KB</td>
</tr>
<tr>
<td>63</td>
<td>PS00</td>
<td>1KB</td>
</tr>
</tbody>
</table>

### 6.10.3.4 Embedded Page Table Configuration Register (EPTCFG)

This read-only register consists of 3 pairs of page size (PSi) and sub-page size (SPSi) values, where i = 0 to 2. These combinations are supported for Page Table translations. The page size and sub-page size encodings for PSi and SPSi are the same the MAS1FSIZE encodings, except that an SPSi value of 0b00001 is reserved and a value of zero for the SPSi field means there is no page size and sub-page size combination information supplied by that field. If SPSi is zero, PSi is zero. Zero values of PSi and SPSi pairs are the leftmost fields. See Table 3. For nonzero values of SPSi, PSi minus SPSi is greater than 7.

The EPTCFG register is a privileged register except that if the Embedded.Hypervisor category is supported,
EPTCFG register is a hypervisor resource. The layout of the EPTCFG register is shown in Figure 33.

```
   //    PS2   SPS2   PS1   SPS1   PS0   SPS0
32  34  39  44  49  54  59  63
```

**Figure 33. Embedded Page Table Configuration Register**

The EPTCFG fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34:38</td>
<td><strong>Page Size 2 (PS2)</strong></td>
</tr>
<tr>
<td></td>
<td>PS2 indicates whether an indirect TLB entry with a page size of $2^{PS2}$ KB combined with the sub-page size specified by SPS2 is supported.</td>
</tr>
<tr>
<td>39:43</td>
<td><strong>Sub-Page Size 2 (SPS2)</strong></td>
</tr>
<tr>
<td></td>
<td>SPS2 indicates whether an indirect TLB entry with a sub-page size of $2^{SPS2}$ KB combined with the page size specified by PS2 is supported.</td>
</tr>
<tr>
<td>44:48</td>
<td><strong>Page Size 1 (PS1)</strong></td>
</tr>
<tr>
<td></td>
<td>PS1 indicates whether an indirect TLB entry with a page size of $2^{PS1}$ KB combined with the sub-page size specified by SPS1 is supported.</td>
</tr>
<tr>
<td>49:53</td>
<td><strong>Sub-Page Size 1 (SPS1)</strong></td>
</tr>
<tr>
<td></td>
<td>SPS1 indicates whether an indirect TLB entry with a sub-page size of $2^{SPS1}$ KB combined with the page size specified by PS1 is supported.</td>
</tr>
<tr>
<td>54:58</td>
<td><strong>Page Size 0 (PS0)</strong></td>
</tr>
<tr>
<td></td>
<td>PS0 indicates whether an indirect TLB entry with a page size of $2^{PS0}$ KB combined with the sub-page size specified by SPS0 is supported.</td>
</tr>
<tr>
<td>59:63</td>
<td><strong>Sub-Page Size 0 (SPS0)</strong></td>
</tr>
<tr>
<td></td>
<td>SPS0 indicates whether an indirect TLB entry with a sub-page size of $2^{SPS0}$ KB combined with the page size specified by PS0 is supported.</td>
</tr>
</tbody>
</table>

### 6.10.3.5 LRAT Configuration Register (LRATCFG) [Category: Embedded.Hypervisor.LRAT]

The LRATCFG read-only register provides configuration information about the LRAT array. LRATCFG is a hypervisor resource. The layout of the LRATCFG registers is shown in Figure 34.

```
   ASSOC  LASIZE  //  LPID  /  NENTRY
   32  40  47  50  51  52  63
```

**Figure 34. LRAT Configuration Register**

The LRATCFG fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:39</td>
<td><strong>Associativity (ASSOC)</strong></td>
</tr>
<tr>
<td></td>
<td>Total number of entries in the LRAT array which can be used for translating addresses with a given LPN. This number is referred to as the associativity level of the LRAT array. A value equal to NENTRY or 0 indicates the array is fully-associative.</td>
</tr>
<tr>
<td>40:46</td>
<td><strong>Logical Address Size (LASIZE)</strong></td>
</tr>
<tr>
<td></td>
<td>Number of bits in a logical address supported by the implementation.</td>
</tr>
<tr>
<td>50</td>
<td><strong>LPID Supported (LPID)</strong></td>
</tr>
<tr>
<td></td>
<td>Indicates whether the LPID field in the LRAT is supported.</td>
</tr>
<tr>
<td></td>
<td>0 The LPID field in the LRAT is not supported.</td>
</tr>
<tr>
<td></td>
<td>1 The LPID field in the LRAT is supported.</td>
</tr>
<tr>
<td>52:63</td>
<td><strong>Number of Entries (NENTRY)</strong></td>
</tr>
<tr>
<td></td>
<td>Number of entries in LRAT array. At least one entry is supported.</td>
</tr>
</tbody>
</table>

All other fields are reserved.

### 6.10.3.6 LRAT Page Size Register (LRATPS) [Category: Embedded.Hypervisor.LRAT]

LRATPS is a read-only register that provides page size information about the LRAT that is implemented if the Embedded.Hypervisor.LRAT category is supported in MMU Architecture Version 2.0. Each Page Size bit (PS0-PS31) that is a one indicates that a specific logical page size is supported by the array. Multiple 1 bits indicate that multiple page sizes are supported concurrently. LRATPS is a hypervisor resource. The layout of the LRATPS registers is shown in Figure 35.

```
   PS31 - PS0
   32  63
```

**Figure 35. LRAT Page Size Register**

The LRATPS fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:63</td>
<td><strong>Page Size 31 - Page Size 0 (PS31-PS0)</strong></td>
</tr>
<tr>
<td></td>
<td>PSm indicates whether a logical page size of $2^m$ KB is supported by the LRAT array. PSm corresponds to bit LRATPS_{64-m} for m = 0 to 31.</td>
</tr>
<tr>
<td></td>
<td>0 Logical page size of $2^m$ KB is not supported.</td>
</tr>
<tr>
<td></td>
<td>1 Logical page size of $2^m$ KB is supported.</td>
</tr>
</tbody>
</table>

All other fields are reserved.
Table 9 on page 1106 shows the relationship between the Page Size (PSm) bits in LRATPS and the logical page size.

6.10.3.7 MMU Control and Status Register (MMUCSR0)

The MMUCSR0 register is used for general control of the MMU including page sizes for programmable fixed size TLB arrays [MAV=1.0] and invalidation of the TLB array. For TLB arrays that have programmable fixed sizes, the TLBn_PS fields [MAV=1.0] allow software to specify the page size. MMUCSR0 is a privileged register except that if the Embedded.Hypervisor category is supported, MMUCSR0 is a hypervisor resource.

The layout of the MMUCSR0 is shown in Figure 36 for MAV=1.0 and in Figure 37 for MAV=2.0.

### Programming Note
Changing the fixed page size of an entire array must be done with great care. If any entries in the array are valid, changing the page size may cause those entries to overlap, creating a serious programming error. It is suggested that the entire TLB array be invalidated and any entries with IPROT have their V bits set to zero before changing page size.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>41:44 TLB3 Array Page Size (TLB3_PS)</td>
<td>Page size of the TLB3 array.</td>
</tr>
<tr>
<td>45:48 TLB2 Array Page Size (TLB2_PS)</td>
<td>Page size of the TLB2 array.</td>
</tr>
<tr>
<td>49:52 TLB1 Array Page Size (TLB1_PS)</td>
<td>Page size of the TLB1 array.</td>
</tr>
<tr>
<td>53:56 TLB0 Array Page Size (TLB0_PS)</td>
<td>Page size of the TLB0 array.</td>
</tr>
</tbody>
</table>

#### Bit Description

- **41:44 TLB3 Array Page Size (TLB3_PS)**
  - A 4-bit field specifies the page size for TLBn array. Page size encoding is defined in Section 6.7.2. If the value of TLBn_PS is not between TLBnCFG_MINSIZE and TLBnCFG_MAXSIZE, the page size is set to TLBnCFG_MINSIZE. A TLBn_PS field is implemented only for a TLB array that can be programmed to support only one of several fixed page sizes. For each TLB array n (for 0 ≤ n < MMUCFG_NTBLS), this field is implemented only if the following are all true.
    - TLBnCFG_AVAIL = 0.
    - TLBnCFG_MINSIZE ≠ TLBnCFG_MAXSIZE.

- **57:62 TLBn Invalidate All**
  - TLB invalidate all bit for the TLBn array.
    - 0 If this bit reads as a 1, an invalidate all operation for the TLBn array is in progress. Hardware will set this bit to 0 when the invalidate all operation is completed. Writing a 0 to this bit during an invalidate all operation is ignored.
    - 1 TLBn invalidation operation. Hardware initiates a TLBn invalidate all operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidate all operation produces an undefined result. If the TLB array supports IPROT, entries that have IPROT set will not be invalidated.

- **57 TLB2 Invalidate All (TLB2_FI)**
  - TLB invalidate all bit for the TLB2 array.
- **58 TLB3 Invalidate All (TLB3_FI)**
  - TLB invalidate all bit for the TLB3 array.
- **61 TLB0 Invalidate All (TLB0_FI)**
  - TLB invalidate all bit for the TLB0 array.
- **62 TLB1 Invalidate All (TLB1_FI)**
  - TLB invalidate all bit for the TLB1 array.

All other fields are reserved.

6.10.3.8 MAS0 Register

The MAS0 register contains fields for identifying and selecting a TLB entry. If the Embedded.Hypervisor.LRAT category is supported, the MAS0 register is also used to select an LRAT entry as well as select between a TLB array and the LRAT array. MAS0 register fields are loaded by the execution of the `tlbsx` instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions.
MAS0 is a privileged register. The layout of the MAS0 register is shown in Figure 38.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Array Type Select (ATSEL) [Category: Embedded.Hypervisor.LRAT]</td>
</tr>
<tr>
<td>33</td>
<td>Selects LRAT or TLB for access for tlbwe and tlbre. In guest state, MAS0ATSEL is treated as if it were zero such that a TLB array is always selected.</td>
</tr>
<tr>
<td>34</td>
<td>TLB Select (TLBSEL)</td>
</tr>
<tr>
<td>35</td>
<td>If ATSEL=0 or MSRGS=1, selects TLB for access. If ATSEL=1, TLBSEL is treated as reserved.</td>
</tr>
<tr>
<td>36</td>
<td>Entry Select (ESEL)</td>
</tr>
<tr>
<td>48</td>
<td>Identifies an entry in the selected array to be used for tlbwe and tlbre. Valid values for ESEL are from 0 to TLBnCFGASSOC - 1 for a TLB array and 0 to LRATCFGASSOC - 1 for the LRAT. That is, ESEL selects the entry in the selected array from the set of entries which can be used for translating addresses with the EPN (if TLBnCFG HES=0 and either MAS0ATSEL=0 or MSRGS=1), the combination of EPN, SIZE, and PID (for tlbwe if TLBnCFG HES=1, and either MAS0ATSEL=0 or MSRGS=1, and for tlbre if TLBnCFG HES=1, and either MAS0ATSEL=0 or MSRGS=0), or the LPN (if MAS0ATSEL=1 and MSRGS=0) specified by MAS2EPN. For fully-associative TLB or LRAT arrays, ESEL ranges from 0 to TLBnCFG NENTRY - 1 or 0 to LRATCFG NENTRY - 1, respectively.</td>
</tr>
<tr>
<td>49</td>
<td>Hardware Entry Select (HES) [MAV=2.0]</td>
</tr>
<tr>
<td>50</td>
<td>Determines how the TLB entry within the selected TLB array is selected by tlbwe if a TLB entry is to be written (MAS0ATSEL=0 or MSRGS=1). If an LRAT entry is to be written (MAS0ATSEL=1 and MSRGS=0) by a tlbwe, HES must be 0. Otherwise the result is undefined. This field has no effect on other TLB Management instructions. Whether an implementation supports this bit for a TLB array is indicated by TLBnCFG HES, and if TLBnCFG HES = 0, HES is ignored and treated as 0 for tlbwe.</td>
</tr>
<tr>
<td>51</td>
<td>Write Qualifier (WQ) [MAV=2.0 and Category: Embedded.TLB Write Conditional]</td>
</tr>
<tr>
<td>52</td>
<td>Qualifies the TLB write operation performed by tlbwe if a TLB entry is to be written (MAS0ATSEL=0 or MSRGS=1). If an LRAT entry is to be written (MAS0ATSEL=1 and MSRGS=0) by a tlbwe and the Embedded.TLB Write Conditional category is supported, WQ must be 0b00. Otherwise the result is undefined. This field has no effect on other TLB Management instructions. Whether an implementation supports this field is indicated by MMUCFGTW C. If MMUCFGTW C = 0, WQ is ignored and treated as 0b00 for tlbwe.</td>
</tr>
<tr>
<td>53</td>
<td>Next Victim (NV)</td>
</tr>
<tr>
<td>54</td>
<td>NV is a hint to software to identify the next victim to be targeted for a TLB miss replacement operation for those TLBs that support the NV function. If the TLB selected by MAS0TLBSEL does not support the NV function, this field is undefined. The method of determining the next victim is implementation-dependent. NV is updated on tlbx hit and miss cases as shown in Table 11 on page 1116, on execution of tlbre if the TLB array being accessed supports the NV field, and on TLB Error interrupts if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor (EPCRDMIUH = 0), or the interrupt is directed to the guest state. When NV is updated by a supported TLB array, the NV field will always present a value that can be used in the MAS0ESEL field. The LRAT array does not support Next Victim.</td>
</tr>
</tbody>
</table>

All other fields are reserved.
### 6.10.3.9 MAS1 Register

The MAS1 register contains fields used for reading and writing an LRAT <E.HVL.RAT> or TLB entry. MAS1 register fields are also loaded by the execution of the `tlbxs` instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions. TLB fields loaded from the MAS1 register are used for selecting a TLB entry during translation. If the Embedded.Hypervisor.LRAT category is supported, LRAT fields `V` and `LSIZE`, which are loaded from `MAS1V` and `TSIZE`, are used for selecting an LRAT entry for translating LPNs when `tlbwx` is executed in guest supervisor state and, if the Embedded.Page Table category is supported, during page table lookups performed when the `PTEREPN` is treated as an LPN (The Embedded.Hypervisor.LRAT category is supported and the TGS bit of the corresponding indirect TLB entry is 1). MAS1 is a privileged register. The layout of the MAS1 register is shown in Figure 39 for MAV=1.0 and in Figure 40 for MAV=2.0.

#### Figure 39. MAS1 register [MAV=1.0]

<table>
<thead>
<tr>
<th>V</th>
<th>PRO</th>
<th>TID</th>
<th>///</th>
<th>TS</th>
<th>TSIZE</th>
<th>///</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>48</td>
<td>51</td>
<td>52</td>
<td>56</td>
</tr>
</tbody>
</table>

#### Figure 40. MAS1 register [MAV=2.0]

<table>
<thead>
<tr>
<th>V</th>
<th>PRO</th>
<th>TID</th>
<th>///</th>
<th>IND</th>
<th>TS</th>
<th>TSIZE</th>
<th>///</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>48</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>57</td>
</tr>
</tbody>
</table>

All other fields are reserved.

### 6.10.3.10 MAS2 Register

The MAS2 register is a 64-bit register which can be read and written as a 64-bit register in 64-bit implementations and as a 32-bit register in 32-bit implementations. In both 64-bit and 32-bit implementations, the MAS2 register contains fields used for reading and writing an LRAT <E.HVL.RAT> or TLB entry. MAS2 register fields are also loaded by the execution of the `tlbxs` instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions. The register contains fields for specifying the effective page address and the storage control attributes for a TLB entry. If the Embedded.Hypervisor.LRAT category is supported, the MAS2 register EPN field can also be used for specifying the logical page number for an LRAT entry. The only MAS2 field used for the LRAT array is EPN. MAS2 is a privileged register. The layout of the MAS2 register is shown in Figure 41 for MAV=1.0 and in Figure 42 for MAV=2.0.

#### Figure 41. MAS2 register [MAV = 1.0]

<table>
<thead>
<tr>
<th>MAS2U</th>
<th>///</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPN</td>
<td>///</td>
</tr>
<tr>
<td>0:32</td>
<td>52</td>
</tr>
</tbody>
</table>

#### Figure 42. MAS2 register [MAV = 2.0]

<table>
<thead>
<tr>
<th>MAS2U</th>
<th>///</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPN</td>
<td>///</td>
</tr>
<tr>
<td>0:32</td>
<td>54</td>
</tr>
</tbody>
</table>

All other fields are reserved.
tion in Section 6.9. Bits that correspond to an offset within the smallest virtual page implemented need not be implemented. Unimplemented EPN bits are treated as 0s.

57  **Alternate Coherency Mode (ACM)**
See the corresponding TLB bit definition in Section 6.7.1. If ACM is not supported by the implementation, this bit is treated as reserved.

58  **VLE Mode (VLE) [Category: VLE]**
See the corresponding TLB bit definition in Section 6.7.1. If the VLE category is not supported, this bit is treated as reserved.

---

**Programming Note**

Some previous implementations may have a TLB storage bit accessed via this bit position and labeled as X0. Software should not use the presence of this bit (the ability to set to 1 and read a 1) to determine if the implementation supports the Alternate Coherency Mode.

58  **VLE Mode (VLE) [Category: VLE]**
See the corresponding TLB bit definition in Section 6.7.1. If the VLE category is not supported, this bit is treated as reserved.

---

**Programming Note**

Some previous implementations may have a TLB storage bit accessed via this position and labeled as X1. Software should not use the presence of this bit (the ability to set to 1 and read a 1) to determine if the implementation supports the VLE.

59  **Write Through (W)**
See the corresponding TLB bit definition in Section 6.7.1.

60  **Caching Inhibited (I)**
See the corresponding TLB bit definition in Section 6.7.1.

61  **Memory Coherence Required (M)**
See the corresponding TLB bit definition in Section 6.7.1.

62  **Guarded (G)**
See the corresponding TLB bit definition in Section 6.7.1.

63  **Endianness (E)**
See the corresponding TLB bit definition in Section 6.7.1.

All other fields are reserved.

### 6.10.3.11 MAS3 Register

The MAS3 register contains fields used for reading and writing an LRAT <E.HVL RAT> or TLB entry. MAS3 register fields are also loaded by the execution of the `tlbsx` instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions. The MAS3 register contains fields for specifying the real page address, user defined attributes, and the permission attributes for a TLB entry. If the Embedded.Page Table category is supported, MAS3 also contains a field specifying the minimum page size specified by each Page Table Entry that is mapped by the indirect TLB entry. If the Embedded.Hypervisor.LRAT category is supported, the low-order LRPN bits of the LRAT array can be read into or written from MAS3.RPNL by hypervisor software. If the Embedded.Hypervisor.LRAT category is supported, the RPN specified by MAS7 and MAS3 is treated as an LPN for `tlbwe` executed in guest supervisor state (see Section 6.9). MAS3 is a privileged register. If the Embedded.Page Table category is supported, MAS3 has different meanings depending on the MAS1IND value. For MAS1IND = 0, the layout of the MAS3 register is shown in Figure 43 for MAV=1.0 and in Figure 44 for MAV=2.0.

**Figure 43. MAS3 register for MAS1IND=0 [MAV=1.0]**

**Figure 44. MAS3 register for MAS1IND=0 [MAV=2.0]**

For MAS1IND = 1, the layout of the MAS3 register is shown in Figure 45.

---

**Figure 45. MAS3 register for MAS1IND=1 [MAV=2.0 and Category: E.PT]**

The MAS3 fields are described below.

**Bit Description**

32:51  **Real Page Number (bits 32:51)** (RPNL or RPN32:51) [MAV=1.0]
The real page number is formed by the upper n bits of (MAS7.RPNL || MAS3.RPNL), where n = 64 - log<sub>2</sub>(page size in bytes) and page size is specified by MAS1.TSIZE for a `tlbwe` instruction and by the SIZE field of the TLB entry if a TLB entry is being read by a `tlbre` or `tlbsx` instruction. RPN32:31 are accessed through MAS7. RPNL bits corresponding to bits that are not implemented in the RPN field of the TLB are treated as reserved.

32:53  **Real Page Number (bits 32:53)** (RPNL or RPN32:53) [MAV=2.0]
The real page number is formed by the upper n bits of (MAS7.RPNL || MAS3.RPNL), where n = 64 - log<sub>2</sub>(page size in bytes) and page size is...
specified by MAS1\textsubscript{TSIZE} for a \textit{tlbwe} instruction, by the \textit{SIZE} field of the TLB entry if a TLB entry is being read by a \textit{tlbre} or \textit{tlbsx} instruction or by the LSIZE field of the LRAT entry if an LRAT entry is being read by a \textit{tlbre} instruction. RPNL bits corresponding to bits that are not implemented in the RPN field of the TLB are treated as reserved.

54:57 \textbf{User Bits} (U0:U3)
See the corresponding TLB bit definition in Section 6.7.1. If one or more of these bits is not implemented in the TLB, the corresponding MAS3 bit is treated as reserved.

If MAS1\textsubscript{IND} = 0, MAS3\textsubscript{58:63} are defined as follows:

58 \textbf{User State Execute Enable (UX)}
See the corresponding TLB bit definition in Section 6.7.1.

59 \textbf{Supervisor State Execute Enable (SX)}
See the corresponding TLB bit definition in Section 6.7.1.

60 \textbf{User State Write Enable (UW)}
See the corresponding TLB bit definition in Section 6.7.1.

61 \textbf{Supervisor State Write Enable (SW)}
See the corresponding TLB bit definition in Section 6.7.1.

62 \textbf{User State Read Enable (UR)}
See the corresponding TLB bit definition in Section 6.7.1.

63 \textbf{Supervisor State Read Enable (SR)}
See the corresponding TLB bit definition in Section 6.7.1.

If MAS1\textsubscript{IND} = 1, MAS3\textsubscript{58:63} are defined as follows:

58:62 \textbf{Sub-Page Size (SPSIZE)}
See the corresponding TLB field definition in Section 6.7.1.

63 \textbf{Undefined} (UND)
The value of this bit is undefined after a \textit{tlbre} or \textit{tlbsx}.

All other fields are reserved.

6.10.3.12 MAS4 Register
The MAS4 register contains fields for specifying default information to be pre-loaded on an Instruction or Data TLB Error interrupt if the Embedded Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor (EPCR\textsubscript{DMUH} = 0), or the interrupt is directed to the guest state. See Section 6.11.4.7 for more information. MAS4 is a privileged register. The layout of the MAS4 register is shown in Figure 46 for MAV=1.0 and in Figure 47 for MAV=2.0.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure46.png}
\caption{MAS4 register [MAV=1.0]}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure47.png}
\caption{MAS4 register [MAV=2.0]}
\end{figure}

The MAS4 fields are described below.

\begin{itemize}
\item \textbf{Bit Description}
\item 34:35 \textit{TLBSEL Default Value} (TLBSELD) Specifies the default value loaded in MAS0\textsubscript{TLBSEL} on the interrupt.
\item 48 \textit{IND Default Value} (INDD) Specifies the default value loaded in MAS1\textsubscript{IND} and MAS6\textsubscript{SIND} on the interrupt.
\item 52:55 \textit{Default TSIZE Value} (TSIZED) \textbf{[MAV=1.0]} Specifies the default value loaded into MAS1\textsubscript{TSIZE} on a TLB miss exception.
\item 52:56 \textit{Default TSIZE Value} (TSIZED) \textbf{[MAV=2.0]} Specifies the default value loaded into MAS1\textsubscript{TSIZE} on a TLB miss exception. If MMUCFG\textsubscript{TWC} = 1, TSIZED is also the default value loaded into MAS6\textsubscript{ISIZE} on the interrupt.
\item 57 \textit{Default ACM Value} (ACMD) Specifies the default value loaded into MAS2\textsubscript{ACM} on the interrupt.
\item 58 \textit{Default VLE Value} (VLED) \textbf{[Category: VLE]} Specifies the default value loaded into MAS2\textsubscript{VLE} on the interrupt.
\item 59 \textit{Default W Value} (WD) Specifies the default value loaded into MAS2\textsubscript{W} on the interrupt.
\item 60 \textit{Default I Value} (ID) Specifies the default value loaded into MAS2\textsubscript{I} on the interrupt.
\item 61 \textit{Default M Value} (MD) Specifies the default value loaded into MAS2\textsubscript{M} on the interrupt.
\item 62 \textit{Default G Value} (GD) Specifies the default value loaded into MAS2\textsubscript{G} on the interrupt.
\end{itemize}
63  **Default E Value** (ED)
    Specifies the default value loaded into MAS2E on the interrupt.

All other fields are reserved.

### 6.10.3.13 MAS5 Register

The MAS5 register contains fields for specifying LPID and GS values to be used when searching TLB entries with the `tlbsrx`, `<E.TWC>` and `tlbsx` instructions. The SLPID and SGS fields are used to match TLPID and TGS fields in the TLB entry. The MAS5 fields are also used for selecting TLB entries to be invalidated by the `tlbilx` or `tlbivax` instructions. MAS5 is a hypervisor resource. The layout of the MA5 register is shown in Figure 48.

#### Figure 48. MAS5 register

The MAS5 fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td><strong>Search GS</strong> (SGS)</td>
</tr>
<tr>
<td>Specifies the GS value used when searching the TLB during execution of <code>tlbsrx</code>, <code>&lt;E.TWC&gt;</code> and <code>tlbsx</code> and for selecting TLB entries to be invalidated by <code>tlbilx</code> or <code>tlbivax</code>. The SGS field is compared with the TGS field of each TLB entry to find a matching entry.</td>
<td></td>
</tr>
</tbody>
</table>

| 52:63 | **Search Logical Partition ID** (SLPID) |
| Specifies the LPID value used when searching the TLB during execution of `tlbsrx`, `<E.TWC>` and `tlbsx` and for selecting TLB entries to be invalidated by `tlbilx` or `tlbivax`. The SLPID field is compared with the TLPID field of each TLB entry to find a matching entry. Only the least significant MMUCFG.LPID.SIZE bits of SLPID are implemented. |

All other fields are reserved.

#### Programming Note

Hypervisor software should generally treat MAS5 as part of the partition state.

### 6.10.3.14 MAS6 Register

The MAS6 register contains fields for specifying PID, IND, and AS values to be used when searching TLB entries with the `tlbsx` instruction and, if MMUCFG.TWC = 1 or TLBnCFG.HES = 1, for specifying the PID, IND, AS, and size of the virtual address space to be used for selecting TLB entries to be invalidated by the `tlbilx` T=3 or `tlbivax` instructions. MAS6 is a privileged register.

#### Figure 49. MAS6 register [MAV = 1.0]

The layout of the MAS6 register is shown in Figure 49 for MAV=1.0 and in Figure 50 for MAV=2.0.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td><strong>Search PID</strong> (SPID)</td>
</tr>
<tr>
<td>Specifies the value of PID used when searching the TLB during execution of <code>tlbsx</code> and for selecting TLB entries to be invalidated by <code>tlbilx</code> T=1 or T=3 and <code>tlbivax</code> with EA61=0. The number of bits implemented is the same as the number of bits implemented in the PID register.</td>
<td></td>
</tr>
</tbody>
</table>

| 52:56 | **Invalidation Size** (ISIZE) [MAV=2.0] |
| ISIZE defines the size of the virtual address space mapped by the TLB entry to be invalidated by `tlbilx` T=3 and `tlbivax`. This field is only supported if MMUCFG.TWC = 1 or, for any TLB array, TLBnCFG.HES = 1. Otherwise this field is reserved. |

| 62 | **Indirect Value for Searches** (SIND) [MAV=2.0 and Category: Embedded.Page Table] |
| Specifies the value of IND used when searching the TLB during execution of `tlbsx`. It also defines the Indirect (IND) value of the TLB entry to be invalidated by `tlbilx` T=3 and `tlbivax`. |

| 63 | **Address Space Value for Searches** (SAS) |
| Specifies the value of AS used when searching the TLB during execution of `tlbsx`. It also defines the TS value of the TLB entry to be invalidated by `tlbilx` T=3 and `tlbivax`. |

All other fields are reserved.
6.10.3.15 MAS7 Register

The MAS7 register contains a field used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. MAS7 register field is also loaded by the execution of the `tlbsx` instruction. The MAS7 register contains the high order address bits of the RPN for a TLB entry in implementations that support more than 32 bits of physical address. If the Embedded.Hypervisor.LRAT category is supported by such implementations, the high-order LRPN bits of the LRAT array can be read into or written from MAS7 by hypervisor software. If the Embedded.Hypervisor.LRAT category is supported, the RPN specified by MAS7 and MAS3 is treated as an LPN for `tlbwe` executed in guest supervisor state (see Section 6.9). If no more than 32 bits of physical addressing are supported, it is implementation-dependent whether MAS7 is implemented. MAS7 is a privileged register. The layout of the MAS7 is shown in Figure 51.

Figure 51. MAS7 register

The MAS7 fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:63</td>
<td>Real Page Number (bits 0:31) (RPNU or RPN0:31)</td>
</tr>
<tr>
<td>32</td>
<td>Translation Guest State (TGS)</td>
</tr>
</tbody>
</table>

6.10.3.16 MAS8 Register [Category: Embedded.Hypervisor]

The MAS8 register contains fields used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. The MAS8 register fields are also loaded from a matching TLB entry by execution of a `tlbsx` instruction that is successful. The associated TLB fields are used to select a TLB entry during TLB address translation and TLB searches, and to force a Data Storage Interrupt to be directed to the hypervisor state. MAS8 is a hypervisor resource. The LPID field of the LRAT is used to select an LRAT entry during LRAT translation. The layout of the MAS8 register is shown in Figure 52.

Figure 52. MAS8 register

The MAS8 fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Translation Guest State (TGS)</td>
</tr>
</tbody>
</table>

6.10.3.17 Accesses to Paired MAS Registers

In 64-bit implementations, certain MAS registers can be accessed in pairs with a single `mtspr` or `mfspr` instruction. The registers that can be accessed this way are shown in Table 10. These register pairs are treated as if they are a single 64-bit register by a `mtspr` or `mfspr` instruction.

Table 10: MAS Register Pairs

<table>
<thead>
<tr>
<th>64-bit Pairs</th>
<th>SPR Number</th>
<th>Privileged <code>mtspr</code> &amp; <code>mfspr</code></th>
<th>Cat2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASS II MAS6</td>
<td>348</td>
<td>hypv&lt;sup&gt;1&lt;/sup&gt;</td>
<td>E.HV; 64</td>
</tr>
<tr>
<td>MASS II MAS1</td>
<td>349</td>
<td>hypv&lt;sup&gt;1&lt;/sup&gt;</td>
<td>E.HV; 64</td>
</tr>
<tr>
<td>MASS7 II MAS3</td>
<td>372</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>MASS0 II MAS1</td>
<td>373</td>
<td>yes</td>
<td>64</td>
</tr>
</tbody>
</table>

1 This register is a hypervisor resource, and can be accessed by one of these instructions only in hypervisor state (see Chapter 2).
2 See Section 1.3.5 of Book I. If multiple categories are listed, the register pair is only provided if all categories are supported. Otherwise the SPR number is treated as reserved.

Programming Note

Hypervisor software should generally treat MAS8 as part of the partition state. After executing `tlbsx` and `tlbre`, hypervisor software may need to restore MAS8 before returning to guest state. This is especially important if the Embedded.Hypervisor.LRAT category is supported because a guest can execute a `tlbwe` instruction that writes a TLB entry with the MAS8 values.

Programming Note

For a TLB entry with VF=1, hypervisor software should have the execution permission bits set so that an instruction fetch of the page is prevented.

The VF bit can be used to force a Data Storage interrupt for virtualization of MMIO.
6.10.3.18 MAS Register Update Summary

Table 11 summarizes how MAS registers are modified by Instruction TLB Error interrupt, Data TLB Error interrupt and the TLB Management instructions.
### Table 11: MAS Register Update Summary for TLB operations

<table>
<thead>
<tr>
<th>MAS Field Updated</th>
<th>Data TLB Error Interrupt on Load or Store that is not category E.PD or Instruction TLB Error Interrupt&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Data TLB Error Interrupt on External Process ID Load &lt;E.PD&gt;&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Data TLB Error Interrupt on External Process ID Store &lt;E.PD&gt;&lt;sup&gt;2&lt;/sup&gt;</th>
<th>$t_{lsx}$ hit</th>
<th>$t_{lsx}$ miss</th>
<th>$t_{ibre}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAS0&lt;sub&gt;ATSEL&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;E.HV&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>MAS0&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>TLB array that hit</td>
<td>MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>—</td>
</tr>
<tr>
<td>MAS0&lt;sub&gt;ESSEL&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then hardware hint, else undefined</td>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then hardware hint, else undefined</td>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then hardware hint, else undefined</td>
<td>index of entry that hit</td>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then hardware hint, else undefined</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>MAS0&lt;sub&gt;HES&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLBnCFG&lt;sub&gt;HES&lt;/sub&gt; for array specified by MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>TLBnCFG&lt;sub&gt;HES&lt;/sub&gt; for array specified by MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>TLBnCFG&lt;sub&gt;HES&lt;/sub&gt; for array specified by MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>0</td>
<td>TLBnCFG&lt;sub&gt;HES&lt;/sub&gt; for array specified by MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>MAS0&lt;sub&gt;WQ&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;E.TWC&gt;</td>
<td>0b01</td>
<td>0b01</td>
<td>0b01</td>
<td>0b01</td>
<td>0b01</td>
<td>—</td>
</tr>
<tr>
<td>MAS0&lt;sub&gt;NV&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then next hardware hint, else undefined</td>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then next hardware hint, else undefined</td>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then next hardware hint, else undefined</td>
<td>if TLB array with the matching entry supports next victim then next hardware hint, else undefined</td>
<td>if TLB array [MAS4&lt;sub&gt;TLBSEL&lt;/sub&gt;] supports next victim then next hardware hint, else undefined</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>MAS1&lt;sub&gt;V&lt;/sub&gt;</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>TLB&lt;sub&gt;V&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS1&lt;sub&gt;IPROT&lt;/sub&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TLB&lt;sub&gt;IPROT&lt;/sub&gt;</td>
<td>0</td>
<td>TLB&lt;sub&gt;IPROT&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS1&lt;sub&gt;TID&lt;/sub&gt;</td>
<td>PID</td>
<td>EPLC&lt;sub&gt;EPID&lt;/sub&gt;</td>
<td>EPSC&lt;sub&gt;EPID&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;TID&lt;/sub&gt;</td>
<td>MAS6&lt;sub&gt;SPID&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;TID&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS1&lt;sub&gt;IND&lt;/sub&gt;</td>
<td>&lt;E.PT&gt;</td>
<td>MAS4&lt;sub&gt;INDD&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;INDD&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;IND&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;INDD&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;IND&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS1&lt;sub&gt;TTS&lt;/sub&gt;</td>
<td>MSR&lt;sub&gt;RS&lt;/sub&gt; or MSR&lt;sub&gt;DS&lt;/sub&gt;</td>
<td>EPLC&lt;sub&gt;EAS&lt;/sub&gt;</td>
<td>EPSC&lt;sub&gt;EAS&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;TTS&lt;/sub&gt;</td>
<td>MAS6&lt;sub&gt;SAS&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;TTS&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS1&lt;sub&gt;SIZE&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TSIZED&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TSIZED&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TSIZED&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;SIZE&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;TSIZED&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;SIZE&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS2&lt;sub&gt;EPN&lt;/sub&gt;</td>
<td>EA&lt;sub&gt;0:53&lt;sup&gt;1&lt;/sup&gt;&lt;/sub&gt;</td>
<td>EA&lt;sub&gt;0:53&lt;/sub&gt;</td>
<td>EA&lt;sub&gt;0:53&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;EPN&lt;/sub&gt;</td>
<td>undefined</td>
<td>TLB&lt;sub&gt;EPN&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS2&lt;sub&gt;ACM&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;ACMD&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;ACMD&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;ACMD&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;ACM&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;ACMD&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;ACM&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS2&lt;sub&gt;VLE&lt;/sub&gt;</td>
<td>&lt;VLE&gt;</td>
<td>MAS4&lt;sub&gt;VLED&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;VLED&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;VLE&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;VLED&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;VLE&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS2&lt;sub&gt;WIMGE&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;WD ID MD&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;WD ID MD&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;WD ID MD&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;WIMGE&lt;/sub&gt;</td>
<td>MAS4&lt;sub&gt;WD ID MD&lt;/sub&gt;</td>
<td>TLB&lt;sub&gt;WIMGE&lt;/sub&gt;</td>
</tr>
<tr>
<td>MAS3&lt;sub&gt;RPNL&lt;/sub&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TLB&lt;sub&gt;RPN(32:53)&lt;/sub&gt;</td>
<td>0</td>
<td>TLB&lt;sub&gt;RPN(32:53)&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
### Table 11: MAS Register Update Summary for TLB operations

<table>
<thead>
<tr>
<th>MAS Field Updated</th>
<th>Value Loaded on Event</th>
<th>Data TLB Error Interrupt on Load or Store that is not category E.PD or Instruction TLB Error Interrupt²</th>
<th>Data TLB Error Interrupt on External Process ID Load &lt;E.PD&gt;²</th>
<th>Data TLB Error Interrupt on External Process ID Store &lt;E.PD&gt;²</th>
<th>tlbsx hit</th>
<th>tlbsx miss</th>
<th>tlbre</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAS3U0:U3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>if Category: E.PT is not supported or TLBIND = 0 then TLBUX SX UW SW UR SR</td>
<td>0</td>
<td>0</td>
<td>TLBUX SX UW SW UR SR</td>
</tr>
<tr>
<td>MAS3UX SX UW SW UR SR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>if Category: E.PT is not supported or TLBIND = 0 then TLBUX SX UW SW UR SR</td>
<td>0</td>
<td>0</td>
<td>TLBUX SX UW SW UR SR</td>
</tr>
<tr>
<td>MAS3SPSIZE</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>if Category: E.PT supported and TLBIND = 1 then TLB-SPSIZE</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
</tr>
<tr>
<td>MAS3UND</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>If Category: E.PT supported and TLBIND = 1 then undefined</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
<td>(see the entry for MAS3UX SX UW SW UR SR)</td>
</tr>
<tr>
<td>MAS5 &lt;E.HV&gt; &amp; MAS4</td>
<td>—³</td>
<td>—³</td>
<td>—³</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAS6SPID</td>
<td>PID</td>
<td>EPLCEPID</td>
<td>EPSCEPID</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAS6SIZE if TLBnCFGHES=1 or &lt;E.TWC&gt;</td>
<td>MAS4TSIZED</td>
<td>MAS4TSIZED</td>
<td>MAS4TSIZED</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAS6SAS</td>
<td>MSRIS or MRSRS</td>
<td>EPLCEAS</td>
<td>EPSCEAS</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAS6SIND &lt;E.PT&gt;</td>
<td>MAS4INDD</td>
<td>MAS4INDD</td>
<td>MAS4INDD</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAS7RPNU</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TLBRLPN[0:31]</td>
<td>0</td>
<td>TLBRLPN[0:31]</td>
<td>TLBRLPN[0:31]</td>
</tr>
<tr>
<td>MAS8TGS VF TLPID &lt;E.HV&gt;</td>
<td>—³</td>
<td>—³</td>
<td>—³</td>
<td>TLBTGS VF TLPID</td>
<td>—</td>
<td>TLBTGS VF TLPID</td>
<td>—</td>
</tr>
</tbody>
</table>

1. If MSR_CM=0 (32-bit mode) at the time of the exception, EPN0:31 are set to 0.
2. If the E.HV category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor (EPCRDMIUH = 0), or the interrupt is directed to the guest state.
3. MAS5 and MAS8 are not updated on a Data or Instruction TLB Error interrupt. The hypervisor should ensure they already contain values appropriate to the partition.
6.11 Storage Control Instructions

6.11.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a `dcbz` or `dcba` instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 6.6) can cause a delayed Machine Check interrupt or a delayed Check-stop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not retained.

**Data Cache Block Invalidate X-form**

```
dcbi RA,RB
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RA</td>
<td>RB</td>
<td>470</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if RA=0 then b ← 0
else         b ← (RA)
EA ← b + (RB)
InvalidateDataCacheBlock( EA )

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any thread, then the block is invalidated in those data caches. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in any such data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of this thread, then the block is invalidated in that data cache. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in that data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a Store (see Section 6.7.6.5) on implementations that invalidate a block without first writing to main storage all locations in the block that are considered to be modified in the data cache, except that the invalidation is not ordered by `mbar`. On other implementations this instruction is treated as a Load (see the section cited above).

If a thread holds a reservation and some other thread executes a `dcbi` that specifies a location in the same reservation granule, the reservation may be lost only if the `dcbi` is treated as a Store.

`dcbi` may cause a cache locking exception, the details of which are implementation-dependent.

This instruction is privileged.

**Special Registers Altered:**

None
6.11.2 Cache Locking [Category: Embedded Cache Locking]

The Embedded Cache Locking category defines instructions and methods for locking cache blocks for frequently used instructions and data. Cache locking allows software to instruct the cache to keep latency sensitive data readily available for fast access. This is accomplished by marking individual cache blocks as locked.

A locked block differs from a normal block in the cache in the following way:
- Blocks that are locked in the cache do not participate in the normal replacement policy when a block must be replaced.

6.11.2.1 Lock Setting, Query, and Clearing

Cache Locking instructions are used by software to indicate which blocks in a cache should be locked, unlocked, or queried for lock status.

Blocks are locked into the cache by software using Lock Set instructions. The following instructions are provided to lock data items into the data and instruction cache:
- `dcbls` - Data cache block touch and lock set.
- `dcbstls` - Data cache block touch for store and lock set.
- `icbls` - Instruction cache block touch and lock set.

The RA and RB operands in these instructions are used to identify the block to be locked. The CT field indicates which cache in the cache hierarchy should be targeted. These instructions set CR0 to indicate whether the block is locked or is not locked.

The status of whether a cache block is locked in the cache can be queried by software using Lock Query instructions:
- `dcblq` - Data cache block lock query.
- `icblq` - Instruction cache block lock query.

The RA and RB operands in these instructions are used to identify the block to be queried. The CT field indicates which cache in the cache hierarchy should be targeted. These instructions set CR0 to indicate whether the block is locked or is not locked.

An implementation is not required to unlock blocks that contain data that has been invalidated unless it is explicitly unlocked with a `dcblc` or `icblc` instruction; if the implementation does not unlock the block upon invalidation, the block remains locked even though it contains invalid data. If the implementation does not clear locks when the associated block is invalidated, the method of locking is said to be persistent; otherwise it is not persistent. An implementation may choose to implement locks as persistent or not persistent; however, the preferred method is persistent.

It is implementation-dependent if cache blocks are implicitly unlocked in the following ways:
- A locked block is invalidated as the result of a `dcbi`, `dcbf`, `dcbls`, `icbls`, or `icblc` instruction.
- A locked block is evicted because of an overlocking condition.
- A snoop hit on a locked block that requires the block to be invalidated. This can occur because the data the block contains has been modified external to the thread, or another thread has explicitly invalidated the block.
- The entire cache containing the locked block is invalidated.

6.11.2.2 Error Conditions

Setting locks in the cache can fail for a variety of reasons. A Lock Set instruction addressing a byte in storage that is not allowed to be accessed by the storage access control mechanism (see Section 6.7.6) will cause a Data Storage interrupt (DSI). Addresses referenced by Cache Locking instructions are always translated as data references; therefore, `icblts` instructions that fail to translate or are not allowed by the storage access control mechanism cause Data TLB Error interrupts and Data Storage interrupts, respectively. Cache Locking and clearing operations can fail due to non-privileged access. The methods for determining other failure conditions such as unable-to-lock or overlocking (see below), is implementation-dependent.

If the Embedded.Hypervisor category is supported and MSRPUCLEP = 1, an attempt to execute a Cache Locking instruction in guest state results in an Embedded Hypervisor Privilege exception if MSRPR = 0 or a cache...
locking exception if \( \text{MSR}_{\text{PR}} = 1 \). When the Embedded.Hypervisor category is not supported, \( \text{MSR}_{\text{PUCLEP}} = 0 \), or \( \text{MSR}_{\text{GS}} = 0 \), then if a Cache Locking instruction is executed in user mode and \( \text{MSR}_{\text{UCLE}} = 0 \), a cache locking exception occurs. If a Data Storage interrupt occurs as a result of a cache locking exception, one of the following ESR or GESR bits is set to 1 (GESR if the Embedded.Hypervisor category is supported and the interrupt is directed to the guest. Otherwise, ESR).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>DLK(_0)</td>
</tr>
<tr>
<td></td>
<td>0 Default setting.</td>
</tr>
<tr>
<td></td>
<td>1 A \textit{dcbtls}, \textit{dcbtstsl}, \textit{dcblq}, or \textit{dcblc} instruction was executed in user mode.</td>
</tr>
<tr>
<td>43</td>
<td>DLK(_1)</td>
</tr>
<tr>
<td></td>
<td>0 Default setting.</td>
</tr>
<tr>
<td></td>
<td>1 An \textit{icbtls} or \textit{icblc} instruction was executed in user mode.</td>
</tr>
</tbody>
</table>

The behavior of Cache Locking instructions in guest privileged state (\textit{dcbtls}, \textit{dcbtstsl}, \textit{dcblc}, \textit{dcblq}, \textit{icbtls}, \textit{icblc}, \textit{icblq}) is dependent on the setting of \( \text{MSR}_{\text{PUCLEP}} \). When \( \text{MSR}_{\text{PUCLEP}} = 0 \), Cache Locking instructions are permitted to execute normally in the guest privileged state. When \( \text{MSR}_{\text{PUCLEP}} = 1 \), cache locking instructions are not permitted to execute in the guest privileged state and cause an Embedded Hypervisor Privilege exception when executed. See Section 4.2.2, “Machine State Register Protect Register (MSRP)”.

6.11.2.2.1 Overlocking

If no exceptions occur for the execution of an \textit{dcbtls}, \textit{dcbtstsl}, or \textit{icbtls} instruction, an attempt is made to lock the specified block into the cache. If all of the available cache blocks into which the specified block may be loaded are already locked, an overlocking condition occurs. The overlocking condition may be reported in an implementation-dependent manner.

If an overlocking condition occurs, it is implementation-dependent whether the specified block is not locked into the cache or if another locked block is evicted and the specified block is locked.

The selection of which block is replaced in an overlocking situation is implementation-dependent. The overlocking condition is still said to exist, and is reflected in any implementation-dependent overlocking status.

An attempt to lock a block that is already present and valid in the cache will not cause an overlocking condition.

If a cache block is to be loaded because of an instruction other than a Cache Management or Cache Locking instruction and all available blocks into which the block can be loaded are locked, the instruction executes and completes, but no cache blocks are unlocked and the block is not loaded into the cache.

---

**Programming Note**

Since caches may be shared among threads, an overlocking condition may occur when loading a block even though a given thread has not locked all the available cache blocks. Similarly, blocks may be unlocked as a result of invalidations by other threads.

6.11.2.2.2 Unable-to-lock, Unable-to-unlock, and Unable-to-query Conditions

If no exceptions occur and no overlocking condition exists, an attempt to set, query, or unlock a lock may fail if any of the following are true:

- The target address is marked Caching Inhibited, or the storage control attributes of the address use a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.
- Any other implementation-specific error conditions are detected.

If an unable-to-lock or unable-to-unlock condition occurs, the Lock Set or Lock Clear instruction is treated as a no-op and the condition may be reported in an implementation-dependent manner. If an unable-to-query condition occurs, the CR0 status bit for the query is set to 0.
6.11.2.3 Cache Locking Instructions

**Data Cache Block Lock Query X-form**

```
dcblq        CT,RA,RB
```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | /  | CT | RA | RB | 422 | 1 |

Let the effective address (EA) be the sum (RA0)+(RB).
The block containing the byte addressed by EA in the
data cache specified by the CT field is queried to deter-
mine its lock status and CR0 is set as follows:

\[
\text{CR0} = 0b00 \parallel \text{status} \parallel \text{XERSO}
\]

Status is set to 1 if the block is locked in the data cache
specified by the CT field. Status is set to 0 if the block is
not locked in the data cache specified by the CT field.
The instruction is treated as a *Load*.

If an unable-to-query condition occurs (see Section 6.11.2.2.2) status is set to 0.

This instruction is privileged unless the Embedded
Cache Locking,User Mode category is supported. If the
Embedded Cache Locking,User Mode category is sup-
ported, this instruction is privileged only if MSR\textsubscript{UCLE}=0.

**Special Registers Altered:**

CR0

---

**Instruction Cache Block Lock Query X-form**

```
icblq        CT,RA,RB
```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | /  | CT | RA | RB | 198 | 1 |

Let the effective address (EA) be the sum (RA0)+(RB).
The block containing the byte addressed by EA in the
instruction cache specified by the CT field is queried to
determine its lock status and CR0 is set as follows:

\[
\text{CR0} = 0b00 \parallel \text{status} \parallel \text{XERSO}
\]

Status is set to 1 if the block is locked in the instruction
cache specified by the CT field. Status is set to 0 if the
block is not locked in the instruction cache specified by
the CT field.
The instruction is treated as a *Load*.

If an unable-to-query condition occurs (see Section 6.11.2.2.2) status is set to 0.

This instruction is privileged unless the Embedded
Cache Locking,User Mode category is supported. If the
Embedded Cache Locking,User Mode category is sup-
ported, this instruction is privileged only if MSR\textsubscript{UCLE}=0.

**Special Registers Altered:**

CR0
**Data Cache Block Touch and Lock Set**

**X-form**

<table>
<thead>
<tr>
<th>dcbtls</th>
<th>CT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31 / CT</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>RB</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>166</td>
<td>/ 31</td>
</tr>
</tbody>
</table>

Let the effective address (EA) be the sum (RA0)+(RB).

The `dcbtls` instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded and locked into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

If the block already exists in the cache, the block is locked without accessing storage. An unable-to-lock condition may occur (see Section 6.11.2.2.2), or an overlocking condition may occur (see Section 6.11.2.2.1).

The `dcbtls` instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Load*.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR\textsubscript{UCLE}=0.

**Special Registers Altered:**

None

---

**Data Cache Block Touch for Store and Lock Set**

**X-form**

<table>
<thead>
<tr>
<th>dcbtstls</th>
<th>CT, RA, RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31 / CT</td>
</tr>
<tr>
<td>6</td>
<td>RA</td>
</tr>
<tr>
<td>11</td>
<td>RB</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>134</td>
<td>/ 31</td>
</tr>
</tbody>
</table>

Let the effective address (EA) be the sum (RA0)+(RB).

The `dcbtstls` instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded and locked into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

If the block already exists in the cache, the block is locked without accessing storage. An unable-to-lock condition may occur (see Section 6.11.2.2.2), or an overlocking condition may occur (see Section 6.11.2.2.1).

The `dcbtstls` instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Store*.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR\textsubscript{UCLE}=0.

**Special Registers Altered:**

None
**Instruction Cache Block Touch and Lock Set**

`icbtls CT,RA,RB`

Let the effective address (EA) be the sum (RA0)+(RB).

The `icbtls` instruction causes the block containing the byte addressed by EA to be loaded and locked into the instruction cache specified by CT, and provides a hint that the program will probably soon execute code from the block. See Section 4.3 of Book II for a definition of the CT field.

If the block already exists in the cache, the block is locked without refetching from memory.

This instruction treated as a Load (see Section 4.3 of Book II).

If an unable-to-lock condition occurs (see Section 6.11.2.2.2) no operation is performed.

This instruction is privileged unless the Embedded Cache Locking User Mode category is supported. If the Embedded Cache Locking User Mode category is supported, this instruction is privileged only if MSRUCLE=0.

**Special Registers Altered:**

None

**Instruction Cache Block Lock Clear**

`icblc CT,RA,RB`

Let the effective address (EA) be the sum (RA0)+(RB).

The block containing the byte addressed by EA in the instruction cache specified by the CT field is unlocked.

The instruction is treated as a Load.

If an unable-to-lock condition occurs (see Section 6.11.2.2.2) no operation is performed. If the block containing the byte addressed by EA is not locked in the specified cache, no cache operation is performed.

This instruction is privileged unless the Embedded Cache Locking User Mode category is supported. If the Embedded Cache Locking User Mode category is supported, this instruction is privileged only if MSRUCLE=0.

**Special Registers Altered:**

None

**Data Cache Block Lock Clear**

`dcblc CT,RA,RB`

Let the effective address (EA) be the sum (RA0)+(RB).

The block containing the byte addressed by EA in the data cache specified by the CT field is unlocked.

The instruction is treated as a Load.

If an unable-to-lock condition occurs (see Section 6.11.2.2.2) no operation is performed. If the block containing the byte addressed by EA is not locked in the specified cache, no cache operation is performed.

This instruction is privileged unless the Embedded Cache Locking User Mode category is supported. If the Embedded Cache Locking User Mode category is supported, this instruction is privileged only if MSRUCLE=0.

**Special Registers Altered:**

None

**Programming Note**

The `dcblc` and `icblc` instructions are used to remove locks previously set by the corresponding lock set instructions.
6.11.3 Synchronize Instruction

The Synchronize instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system programmers.

When \( L=1 \), the Sync instruction provides an ordering function for the operations caused by the Message Send instruction and previous Stores for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The stores must be performed with respect to the thread receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt is taken.

In conjunction with the `tlbie` and `tlbsync` instructions, when \( L=0 \), the Sync instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the `tlbsync` instruction description on page 1141.

When \( L=0 \), the Sync instruction also provides an ordering function for the operations caused by the Message Send instruction and previous Stores. The stores must be performed with respect to the thread receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt is taken.

6.11.4 LRAT [Category: Embedded.Hypervisor.LRAT] and TLB Management

Unless the Embedded.Page Table category is supported, no format for the Page Tables or the Page Table Entries is implied. Software has significant flexibility in implementing a custom replacement strategy. For example, software may choose to set IPROT=1 for TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB and TLB Miss exceptions to those pages never occur. At a minimum, software must maintain a TLB entry or entries for the Instruction and Data TLB Error interrupt handlers.

TLB management is performed in software with some hardware assist. This hardware assist consists of a minimum of:

- Automatic recording of the effective address causing a TLB Error interrupt. For Instruction TLB Error interrupts, the address is saved in the Save/Restore Register 0. For Data TLB Error interrupts, the address is saved in the Data Exception Address Register.

- Automatic updating of the MAS register on the occurrence of a TLB Error interrupt if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor (EPCRDMUIH = 0), or the interrupt is directed to the guest state.

- Instructions for reading, writing, searching, invalidating, and synchronizing the TLB. If the Embedded.Hypervisor.LRAT category is supported, a subset of these instructions can also be used for reading and writing the LRAT.

  **Programming Note**

If the Embedded.Hypervisor category is supported and if EPCR_TLBGS DTLBGS = 0b00, the hypervisor can virtualize the physical TLB by keeping a software copy of at least the guest operating system TLB entries with IPROT=1 and avoid keeping the guest Instruction and Data TLB Error interrupt handlers in the physical TLB.

6.11.4.1 Reading TLB or LRAT Entries

TLB entries can be read by executing `tlbre` instructions. If the Embedded.Hypervisor.LRAT category is supported, LRAT entries can be read by also executing `tlb` instructions. At the time of `tlbre` execution, a TLB array is selected if \( \text{MAS0_ATSEL} = 0 \) or \( \text{MSRGS} = 1 \). The LRAT array is selected if \( \text{MAS0_ATSEL} = 1 \) and \( \text{MSRGS} = 0 \). The LRAT can be read only when in hypervisor state.

If a TLB array is selected, \( \text{MAS0TLBSEL} \) selects the TLB array to be read. If TLBnCFG_HES=0, the TLB entry in the selected TLB array is selected by \( \text{MAS0ESEL} \) and \( \text{MAS2_EPN} \). In this case, \( \text{MAS0ESEL} \) selects one of the possible entries that can be used for a given \( \text{MAS2_EPN} \). If TLBnCFG_HES=1, the TLB entry in the selected TLB array is selected by \( \text{MAS0ESEL} \) and by a hardware generated hash based on \( \text{MAS1_TID TSIZE} \) and \( \text{MAS2_EPN} \). In this case, \( \text{MAS0ESEL} \) selects one of the possible entries that can be used for a given \( \text{MAS1_TID TSIZE} \) and \( \text{MAS2_EPN} \).

If an LRAT array is selected, the LRAT entry is selected by \( \text{MAS0ESEL} \) and \( \text{MAS2_EPN} \). In this case, \( \text{MAS0ESEL} \) selects one of the possible entries that can be used for a given \( \text{MAS2_EPN} \).

Specifying invalid values for \( \text{MAS0TLBSEL} \), \( \text{MAS0ESEL} \), \( \text{MAS2_EPN} \) or, if TLBnCFG_HES=1, \( \text{MAS1_TID} \) or \( \text{MAS1_TSIZE} \), results in \( \text{MAS1y} \) being set to 0 and undefined results in the other MAS register fields that are loaded by the `tlbre` instruction. Which values are invalid is implementation-dependent. For example, even though an implementation has only one TLB array, the implementation could simply ignore \( \text{MAS0TLBSEL} \), read the selected entry in the TLB array, and load the MAS registers from the TLB entry regardless of the \( \text{MAS0TLBSEL} \) value.
6.11.4.2 Writing TLB or LRAT Entries

TLB entries can be written by executing `tlbwe` instructions. If the Embedded.Hypervisor.LRAT category is supported, LRAT entries can also be written by executing `tlbwe` instructions. At the time of `tlbwe` execution, a TLB array is selected if MAS0ATSEL=0 or MSRGS=1. The LRAT array is selected if MAS0ATSEL=1 and MSRGS=0. The LRAT can be written only when in hypervisor state.

If a TLB array is selected, MAS0TLBSEL selects the TLB array to be written. If TLBnCFGHES=0, the TLB entry in the selected TLB array is selected by MAS0ESEL and MAS2EPN. In this case, MAS0ESEL selects one of the possible entries that can be used for a given MAS2EPN. If TLBnCFGHES=1 and MAS0HES=0, the TLB entry in the selected TLB array is selected by a hardware generated hash based on MAS1TID TSIZE, MAS2EPN, and MAS0ESEL. In this case, MAS0ESEL selects one of the possible entries that can be used for a given MAS0TLSEL, MAS1TID TSIZE, and MAS2EPN. If TLBnCFGHES=1 and MAS0HES=1, the TLB entry in the selected TLB array is selected by a hardware replacement algorithm and a hardware generated hash based on MAS1TID TSIZE and MAS2EPN.

If an LRAT array is selected, the LRAT entry is selected by MAS0ESEL and MAS2EPN. In this case, MAS0ESEL selects one of the possible entries that can be used for a given MAS2EPN. LRAT entries can be written only with MAS0HES=0.

At the time of `tlbwe` execution, the MAS registers contain the contents to be written to the indexed TLB entry. Upon completion of the `tlbwe` instruction, the contents of the MAS registers corresponding to TLB entry fields will be written to the indexed TLB entry, except that if the Embedded.Hypervisor.LRAT category is supported, guest execution of `TLB MANAGEMENT` instructions is enabled (EPCR[GTMI]=0), MSRPR = 0, MSRGS = 1, and, for the TLB array to be written, TLBnCFGGTWE = 1, the RPN from the MAS registers is treated like an LPN and translated by the LRAT, and the RPN from the LRAT is written to the TLB entry if the translation is successful. See Section 6.9. If the LRAT translation fails, an LRAT Miss exception occurs.

If the Embedded.Hypervisor.LRAT category is supported and a guest supervisor executes a `tlbwe` instruction with IPROT=1 or for which the entry to be overwritten has IPROT=1, an Embedded Hypervisor Privilege exception occurs. However, if MAS0WO=0b01, it is implementation-dependent whether the Embedded Hypervisor Privilege exception occurs.

If the Embedded.Hypervisor.LRAT category is supported and a guest supervisor executes a `tlbwe` instruction with MAS0WO=0, it is implementation-dependent whether the Embedded Hypervisor Privilege exception occurs.

If a TLB entry is being written with MAS0HES=1, the hardware replacement algorithm picks an entry in the selected array from the set of entries which can be used for translating addresses with the specified TID, TSIZE and EPN. Whenever possible, an entry with IPROT equal to 0 is selected. However, an Embedded Hypervisor Privilege exception occurs on a `tlbwe` if all the following conditions are met.

- The Embedded.Hypervisor.LRAT category is supported.
- The `tlbwe` is executed in guest supervisor state.
- IPROT=1 for all entries which can be used for translating addresses with the specified TID, TSIZE and EPN.
- MAS0WO= 0b00

If MAS0WO= 0b01, MAS0HES =1, and the first three of the above conditions are met, it is implementation-dependent whether an Embedded Hypervisor Privilege exception occurs.

For TLBs with TLBnCFGHES=1, the relationship between the TLB entry selected by a `tlbwe` with MAS0HES=0 versus the TLB entry selected by a `tlbwe` with MAS0HES=1 is implementation-dependent and may depend on the history of TLB use.

If an invalid value is specified for MAS0TLBSEL, MAS0ESEL or MAS2EPN, either no TLB entry is written by the `tlbwe`, or the `tlbwe` is performed as if some implementation-dependent, valid value were substituted for the invalid value, or an Illegal Instruction exception occurs. If the page size specified by MAS1TSIZE is not supported by the specified array, the `tlbwe` may be performed as if TSIZE were some implementation-dependent value, or an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported but the Embedded.Hypervisor.LRAT category is not supported, the `tlbwe` instruction is hypervisor privileged. Otherwise, this instruction is privileged.

--- Programming Note ---

When reading TLB entries, MAS2EPN or a subset of the bits in MAS2EPN is used to form the index for accessing the TLB array, i.e. MAS2EPN isn’t necessarily an effective page number.

--- Programming Note ---

Since a hardware replacement algorithm selects the entry for a `tlbwe` instruction with MAS0HES =1, it is typically not possible to write the same entry using a second `tlbwe` instruction with MAS0HES =1. Doing so might create multiple entries for the same virtual page. If software needs to change the value of any of the TLB fields, software should generally invalidate the original entry before executing the second `tlbwe` instruction with the new values.
6.11.4.2.1 TLB Write Conditional [Embedded.TLB Write Conditional]

The \textit{tlbsrx}. \textit{<E.TWC>} instruction and \textit{tlbwe} instruction with \textit{MAS0\textsubscript{WO}} = 0b01 together permit a convenient way for software to write a TLB entry while ensuring that the entry is not a duplicate entry and is not a stale, invalid value. Without the TLB Write Conditional facility, software must hold a software lock during the process of creating a TLB entry in order to prevent other threads from updating a shared TLB or invalidating a TLB entry. The \textit{tlbsrx}. \textit{<E.TWC>} instruction has two effects that occur either at the same time or in the following order.

1. A TLB-reservation is established for a virtual address, and, if the Embedded Page Table category is supported, an associated IND value.

2. A search of the selected TLB array is performed. The TLB-reservation is used by a subsequent \textit{tlbwe} instruction that writes a TLB entry (i.e. \textit{MAS0\textsubscript{ATSEL}} = 0 or \textit{MSR\textsubscript{GS}}=1) with \textit{MAS0\textsubscript{WO}} = 0b01. The TLB is only written by this \textit{tlbwe} if the TLB-reservation still exists at the instant the TLB is written. A \textit{tlbwe} that writes the TLB is said to "succeed". TLB Write Conditional cannot be used for the LRAT.

\textbf{TLB-reservation}

A TLB-reservation is set by a \textit{tlbsrx}. \textit{<E.TWC>} instruction. The TLB-reservation has an associated IND \textit{<E.PT>} and virtual address consisting of AS, PID, EA\textsubscript{0:n-1}, LPID \textit{<E.HV>}, and GS \textit{<E.HV>}. These values come from \textit{MAS1\textsubscript{IND} \textit{<E.PT>}; MAS1\textsubscript{TS} MAS1\textsubscript{TID}; MAS2\textsubscript{EPN}; MAS5\textsubscript{SLPID} \textit{<E.HV>}, and MAS5\textsubscript{SGS} \textit{<E.HV>}, respectively. There is no specific page size associated with the TLB-reservation. The TLB-reservation applies to any virtual page that contains the virtual address. There is only one TLB-reservation in a thread.

The TLB-reservation is cleared by any of the following.

- The thread holding the TLB-reservation executes another \textit{tlbsrx}. \textit{<E.TWC>}: This clears the first TLB-reservation and establishes a new one.
- A \textit{tlbivax} is executed by any thread and all the following conditions are met.
  - Either the Embedded.Hypervisor category is not supported or the \textit{MAS0\textsubscript{SGS}} and \textit{MAS0\textsubscript{SLPID}} values used by the \textit{tlbivax} match the GS and LPID values associated with the TLB-reservation.
  - The \textit{MAS5\textsubscript{SLPID}} and \textit{MAS6\textsubscript{SAS}} values used by the \textit{tlbivax} match the GS, LPID, PID, and AS values associated with the TLB-reservation, and a \textit{tlbwe} with \textit{T} = 3 is executed by the thread holding the TLB-reservation.
  - The \textit{EA\textsubscript{0:n-1}} values of the \textit{tlbivax} match the \textit{EA\textsubscript{0:n-1}} values associated with the TLB-reservation, where \textit{n}=$64\cdot\log_2$\textit{(page size in bytes)} and \textit{page size} is specified by the \textit{MAS6\textsubscript{SIZE}}.

A \textit{tlbwe} instruction is executed by the thread holding the TLB-reservation or by a thread that shares the TLB with this thread, and all the following are true.

- An interrupt does not occur as a result of the \textit{tlbwe} instruction.
- The Embedded.Hypervisor category is not supported or the \textit{MAS8\textsubscript{TLPID}} value used by the \textit{tlbwe} matches the LPID value associated with the TLB-reservation.
- The Embedded.Hypervisor category is not supported or the \textit{MAS8\textsubscript{TGS}} value used by the \textit{tlbwe} matches the GS value associated with the TLB-reservation.
- The \textit{MAS1\textsubscript{TID}} value used by the \textit{tlbwe} matches the PID value associated with the TLB-reservation.
- The Embedded.Page Table category is not supported or the \textit{MAS1\textsubscript{IND}} value used by the \textit{tlbwe} matches the IND value associated with the TLB-reservation.
- The \textit{MAS1\textsubscript{TS}} value used by the \textit{tlbwe} matches the AS value associated with the TLB-reservation.
- \textit{Bits 0:(n-1)} of \textit{MAS2\textsubscript{EPN}} used by the \textit{tlbwe} match the \textit{EA\textsubscript{0:n-1}} values associated with the TLB-reservation, where \textit{n}=$64\cdot\log_2$\textit{(page size in bytes)} and \textit{page size} is specified by the \textit{MAS1\textsubscript{SIZE}} used by the \textit{tlbwe}.

Either of the following conditions are met.

- The \textit{MAS0\textsubscript{WO}} used by the \textit{tlbwe} instruction is 0b00.
The MAS0\textsubscript{WO} used by the `tlbwe` instruction is 0b01 and the TLB-reservation for the thread executing the `tlbwe` exists.

The thread that has the TLB-reservation or another thread that shares the TLB with this thread that, as a result of a Page Table translation, writes a TLB entry and all the following conditions are met.

- The TS and EA\textsubscript{0:n-1} values for the new TLB entry match the corresponding values associated with the TLB-reservation where n = 64-log\textsubscript{2}(\textit{page size in bytes}), where \textit{page size} is specified by the SIZE value written to the TLB entry.
- The Embedded.Hypervisor category is not supported or TLPID for the new TLB entry matches the LPID associated with the TLB-reservation.
- The Embedded.Hypervisor category is not supported or TGS for the new TLB entry matches the GS associated with the TLB-reservation.
- The TID for the new TLB entry matches the PID associated with the TLB-reservation.
- The Valid bit for the new TLB entry is 1.
- The IND value associated with the TLB-reservation is 0.

Implementations are allowed to clear a TLB-reservation for conditions other than those specified above. The architecture assures that a TLB-reservation will be cleared when required per the above requirements, but does not guarantee that these are the only conditions for clearing a TLB-reservation. However, the occurrence of an interrupt does not clear a TLB-reservation.

### Programming Note
Software running on two threads that share a TLB should not attempt to create two TLB entries that would both translate a specific virtual address and where the TID or LPID values differ, i.e. one of the values is zero and the other is nonzero. The TLB-reservation will not protect against this case, since a TLB-reservation is not cleared by a `tlbwe` unless there is an exact match on the PID and LPID values.

Likewise software should not attempt to create a Page Table entry and a TLB entry where both entries would translate a specific virtual address, where the TLB array written by the Page Table translation is used by the same thread that uses this TLB entry, and where the TID or LPID values of the PTE and TLB entry differ, i.e. one of the values is zero and the other is nonzero. The TLB-reservation will not protect against this case, since a TLB-reservation is not cleared by a TLB write resulting from a Page Table translation unless there is an exact match on the PID and LPID values.

## Synchronization of TLB-reservation
The side-effect of a `tlbsrx`. `<E.TWC>` instruction setting the TLB-reservation can be synchronized by a context synchronizing instruction or event.

### Programming Note
A common operation is to ensure that the TLB-reservation has been set by a `tlbsrx`. `<E.TWC>` instruction before executing a subsequent \textit{Load} instruction of a software page table entry in order to ensure the TLB-reservation detects an invalidation of the entry that was accessed. Beside using a context synchronizing instruction, software can also ensure the TLB-reservation has been set by a `tlbsrx`. `<E.TWC>` instruction by reading the CR0 field or CR with a `mfocr` or `mfcr` instruction after the `tlbsrx`. `<E.TWC>` and creating a dependency between the data read from CR0 or CR and the address used for the subsequent \textit{Load} instruction.

## Serialization of TLB operations
Regardless of which threads initiated the operations, all operations (reads, writes, invalidates, and searches) involving a single TLB are defined to be serialized such that only one operation occurs at a time. This operation is consistent with the program order of the thread performing the TLB operation. This also applies to a TLB that is shared by multiple threads. Even if there is no matching TLB entry on a `tlbivax`, the TLB is still searched to determine there is no matching entry and this search is still referred to as the TLB invalidation.

If two threads share a TLB and both simultaneously execute a `tlbsrx`. `<E.TWC>` instruction for a virtual address in a virtual page \( V \), and then both threads execute a `TLB Write Conditional` to create a TLB entry for the virtual page \( V \), at most one of these `tlbwe` instructions succeeds.

If, after thread \( P1 \) establishes a TLB-reservation for a virtual address in a virtual page \( V \), another thread \( P2 \) executes a `tlbivax` that invalidates a TLB entry for the virtual page \( V \) and thread \( P1 \) does a `TLB Write Conditional` to create a TLB entry for the virtual page \( V \), then one of the following occurs.

- The TLB invalidation occurs before the TLB write. Thus the TLB-reservation is lost and the `TLB Write Conditional` does not succeed.
- The TLB write occurs before the TLB invalidation. Thus the `TLB Write Conditional` succeeds and the resulting TLB entry created by the `tlbwe` is invalidated by the `tlbivax`.

### Forward progress
Forward progress in loops that use `tlbsrx`. `<E.TWC>` and `tlbwe` with MAS0\textsubscript{WO} = 0b01 is achieved by a cooperative effort among hardware and system software.
The architecture guarantees that when a thread executes a *tibivr* <E.TWC> to set a TLB-reservation for virtual address X and then a *TLB Write Conditional* to write a TLB entry, either

1. the *TLB Write Conditional* succeeds and the TLB entry is written, or
2. the *TLB Write Conditional* fails because the TLB-reservation was reset because some other thread invalidated all TLB entries in the system for the virtual page containing the virtual address X or some other thread wrote a shared TLB entry for the virtual page containing the virtual address X, or
3. the *TLB Write Conditional* fails because the thread's TLB-reservation was lost for some other reason.

In Case 1 forward progress is made in the sense that the thread successfully wrote the TLB entry. In Case 2, the system as a whole makes progress in the sense that either some thread successfully invalidated TLB entries for virtual address X or some thread that shares the TLB wrote a TLB entry for the virtual page containing virtual address X. Case 3 covers TLB-reservation loss required for correct operation of the rest of the system. This includes TLB-reservation loss caused by some other thread invalidating all entries in a shared TLB, as well as TLB-reservation loss caused by system software invalidating all entries for the PID value associated with virtual address X. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of TLB-reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

*Programming Note*

The architecture does not include a “fairness guarantee”. In competing for a TLB-reservation, two threads can indefinitely lock out a third.

### 6.11.4.3 Invalidating TLB Entries

TLB entries may be invalidated by three different methods or if the Embedded.Hypervisor category is supported, by four different methods.

- The TLB entry can be invalidated as the result of a *tibivr* instruction that sets the MAS1γ bit in the entry to 0.
- TLB entries may be invalidated as a result of a *tibiva* instruction or from an invalidation resulting from a *tibiva* on another thread.
- TLB entries may be invalidated as a result of an invalidate all operation specified through appropriate settings in the MMUCSR0.
- If the Embedded.Hypervisor category is supported, TLB entries may be invalidated as a result of a *tibiva* instruction.

See Section 6.11.4.4 for the effects of the above methods on TLB lookaside information.

In systems consisting of a single-threaded processor as well as in systems consisting of multi-threaded processors, invalidations can occur on a wider set of TLB entries than intended. That is, a virtual address presented for invalidation may cause not only the intended TLB targeted for invalidation to be invalidated, but may also invalidate other TLB entries depending on the implementation. This is because parts of the translation mechanism may not be fully specified to the hardware at invalidate time. This is especially true in SMP systems, where the invalidation address must be supplied to all threads in the system, and there may be other limitations imposed by the hardware implementation. This phenomenon is known as generous invalidates. The architecture assures that the intended TLB will be invalidated, but does not guarantee that it will be the only one. A TLB entry invalidated by writing the V bit of the TLB entry to 0 by use of a *tlbwe* instruction is guaranteed to invalidate only the selected TLB entry. Invalidations occurring from *tibiva* or *tibiva* instructions or from *tibiva* instructions on another thread may cause generous invalidates.

The architecture provides a method to protect against generous invalidations. This is important since there are certain virtual memory regions that must be properly mapped to make forward progress. To prevent this, the architecture specifies an IPROT bit for TLB entries. If the IPROT bit is set to 1 in a given TLB entry, that entry is protected from invalidations resulting from *tibiva* <E.HV> and *tibiva* instructions, or from invalidate all operations. TLB entries with the IPROT field set may only be invalidated by explicitly writing the TLB entry and specifying a 0 for the V (MAS1γ) field. This does not preclude the possibility that a TLB entry with the IPROT field set can then invalidate the replaced TLB entry.

To invalidate one or more individual virtual pages from all TLB arrays in all threads without the involvement of software running on other threads, software can execute the following sequence of instructions.

- one or more *tibiva* instructions
- *mbar* or *sync*
- *tibiva* or *tibiva*
Other instructions, excluding \texttt{tlbivax}, may be interleaved with the instruction sequence shown above, but the instructions in the sequence must appear in the order shown. On systems consisting of only a single-threaded processor or on systems where every thread shares every TLB, the \texttt{tlbsync} and the preceding \texttt{mbar} or \texttt{sync} can be omitted.

For the preceding instruction sequence, the \texttt{mbar} or first \texttt{sync} instruction prevents the reordering of \texttt{tlbivax} instructions previously executed by the thread with respect to the subsequent \texttt{tlbsync} instruction. The \texttt{tlbsync} instruction and the subsequent \texttt{sync} instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required and Alternate Coherence Mode attributes, before any data accesses caused by instructions following the \texttt{sync} instruction are performed with respect to that thread or mechanism.

The most obvious issue with generous invalidations is the code memory region that serves as the exception handler for MMU faults. If this region does not have a valid mapping, an MMU exception cannot be handled because the first address of the exception handler will result in another MMU exception.

Not all TLB arrays in a given implementation will implement the IPROT attribute. It is likely that implementations that are suitable for demand page environments will implement it for only a single array, while not implementing it for other TLB arrays.

Operating systems and hypervisors need to use great care when using protected (IPROT) TLB entries, particularly in SMP systems. A system that contains TLB entries on other threads will require a cross thread interrupt or some other synchronization mechanism to assure that each thread performs the required invalidation by writing its own TLB entries.

For MMU Architecture Version 1.0, to ensure a TLB entry that is not protected by IPROT is invalidated if software does not know which TLB array the entry is in, software should issue a \texttt{tlbivax} instruction targeting each TLB in the implementation with the EA to be invalidated.

The preferred method of invalidating entire TLB arrays is invalidation using MMUCSR0, however \texttt{tlbilx} may be more efficient.

Invalidations using MMUCSR0 only affect the TLB array on the thread that performs the invalidation. To perform invalidations on all threads in a coherence domain on a multi-threaded processor or on a system containing multiple single-threaded processors, software should use \texttt{tlbivax}. If a large number of TLB entries need to be invalidated, using MMUCSR0 or, if the Embedded.Hypervisor category is supported, \texttt{tlbilx}, on each thread may be more efficient.

Since a hardware replacement algorithm selects the entry for a \texttt{tlbwe} instruction with \texttt{MAS0HES} = 1, it is typically not possible to invalidate the entry using a second \texttt{tlbwe} instruction with \texttt{MAS0HES} = 1 and \texttt{MAS1V} = 0. If software needs to invalidate a single entry that was written with \texttt{MAS0HES} = 1, software should generally invalidate the entry using \texttt{tlbilx} with T=3 or \texttt{tlbivax}.

6.11.4.4 TLB Lookaside Information

For performance reasons, most implementations also have implementation-specific lookaside information that is used in address translation. This lookaside information is a cache of recently used TLB entries.

If TLBnCFG\_HES=0, lookaside information for the associated TLB array is kept coherent with the TLB and is invisible to software. Any write to the TLB array that displaces or updates an entry will be reflected in the
lookaside information, invalidating the lookaside information corresponding to the previous TLB entry. Any type of invalidation of an entry in TLB will also invalidate the corresponding entry in the lookaside information.

If TLBnCFG_HES=1, lookaside information for the associated TLB array is not required to be kept coherent with the TLB. Only in the following conditions will the lookaside information be kept coherent with the TLB. The MMUCR0 TLB invalidate all will invalidate all lookaside information. The \texttt{tlbilx} and \texttt{tlbivax} instructions invalidate lookaside information corresponding to TLB entry values that are specified to invalidate as well as those TLB entry values that would have been invalidated except for their IPROT=1 value.

The same instructions that synchronize invalidations of TLB entries also synchronize invalidation of TLB lookaside information.

\textbf{Programming Note}

If TLBnCFG_HES=1 for a TLB array and it is important that the lookaside information corresponding to a TLB entry be invalidated, software should use \texttt{tlbilx} or \texttt{tlbivax} to invalidate the virtual address.

6.11.4.5 Invalidating LRAT Entries

There is only one mechanism for invalidating LRAT entries. An LRAT entry can be invalidated as the result of a \texttt{tlbwe} instruction that overwrites the LRAT entry with a new valid entry or that sets LRAT_V = 0. Only one LRAT entry is invalidated by a single \texttt{tlbwe}.

6.11.4.6 Searching TLB Entries

Software may search the MMU by using the \texttt{tlbsx} instruction, and, if Category: TLB Write Conditional category is supported, the \texttt{tlbsrx}. \langle E.TWC \rangle instruction. The \texttt{tlbsrx}, \langle E.TWC \rangle and \texttt{tlbsx} instructions use IND, PID, and AS values from the MAS registers instead of the PID registers and the MSR, and, if the Embedded.Hypervisor category is supported, these instructions use an LPID and GS value from the MAS registers instead LPIDR and MSR. This allows software to search address spaces that differ from the current address space defined by the PID registers. This is useful for TLB fault handling.

6.11.4.7 TLB Replacement Hardware Assist

The architecture provides mechanisms to assist software in creating and updating TLB entries when certain MMU related exceptions occur. This is called TLB Replacement Hardware Assist. Hardware will update the MAS registers on the occurrence of a Data TLB Error Interrupt or Instruction TLB Error interrupt if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor (EPCRDMIUH = 0), or the interrupt is directed to the guest state.

When a Data or Instruction TLB Error interrupt (TLB miss) occurs and if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor (EPCRDMIUH = 0), or the interrupt is directed to the guest state, then MAS0, MAS1, and MAS2 are automatically updated using the defaults specified in MAS4 as well as the AS and EPN values corresponding to the access that caused the exception. MAS6 is updated to set MAS6SPID to the value of PID (EPLC\_EPID for External PID Load instructions or EPSC\_EPID for External PID Store instructions), MAS6SIND to the value of MAS4INDD, and MAS6SAS to the value of MSR\_DS or MSR\_IS depending on the type of access (data or instruction) that caused the error. In addition, if MAS4TLBSELD identifies a TLB array that supports NV (Next Victim), MAS0ESEL is loaded with a value that hardware predicts represents the best TLB entry to victimize to create a new TLB entry and MAS0NV is updated with the TLB entry index of what hardware predicts to be the next victim for the set of entries which can be used for translating addresses with the EPN that caused the exception. Thus MAS0ESEL identifies the current TLB entry to be replaced, and MAS0NV points to the next victim. When software writes the TLB entry, the MAS0NV field is written to the TLB array’s set of next victim values. The algorithm used by the hardware to determine which TLB entry should be targeted for replacement is implementation-dependent.

Next Victim support is provided for TLB arrays that are set associative and that have TLBnCFG_HES=0. Next Victim support is not provided for TLB arrays that are fully associative.

The automatic update of the MAS registers sets up all the necessary fields for creating a new TLB entry with the exception of RPN, the U0-U3 attribute bits, and the permission bits. With the exception of the upper 32 bits of RPN and the page attributes (should software desire to specify changes from the default attributes), all the remaining fields are located in MAS3, requiring only the single MAS register manipulation by software before writing the TLB entry.

For Instruction Storage interrupt (ISI) and Data Storage interrupt (DSI) related exceptions, the MAS registers are not updated. Software must explicitly search the TLB to find the appropriate entry.

The update of MAS registers through TLB Replacement Hardware Assist is summarized in Table 11 on page 1116.
6.11.4.8 32-bit and 64-bit Specific MMU Behavior

MMU behavior is largely unaffected by whether the thread is in 32-bit computation mode (MSR\textsubscript{CM}=0) or 64-bit computation mode (MSR\textsubscript{CM}=1). The only differences occur in the EPN field of the TLB entry and the EPN field of MAS2. The differences are summarized here.

- Executing a \texttt{tlbwe} instruction in 32-bit mode will set bits 0:31 of the TLB EPN field to zero unless \texttt{MAS0ATSEL} is set, in which case those bits are not written to zero.
- For an update to MAS registers via TLB Replacement Hardware Assist (see Section 6.11.4.7), an update to bits 0:53 of the EPN field occurs regardless of the computation mode of the thread at the time of the exception or the interrupt computation mode in which the interrupt is taken. If the instruction causing the exception was executing in 32-bit mode, bits 0:31 of the EPN field in MAS2 will be set to 0.
- Executing a \texttt{tlbre} instruction in 32-bit mode will set bits 0:31 of the MAS2 EPN field to an undefined value.
- In 32-bit implementations, MAS2U can be used to read or write EPN\textsubscript{0:31} of MAS2.

\textbf{Programming Note}

This allows a 32-bit OS to operate seamlessly in 32-bit mode on a 64-bit implementation and a 64-bit OS to easily support 32-bit applications.
6.11.4.9 TLB Management Instructions

The `tlbivax` instruction is used to invalidate TLB entries. Additional instructions are used to read and write, and search TLB entries, and to provide an ordering function for the effects of `tlbivax`. If the Embedded.Hypervisor category is supported, the `tlbilx` instruction is used to invalidate TLB entries in the thread executing the `tlbilx`.

**TLB Invalidate Virtual Address Indexed X-form**

```
tlbivax RA, RB
```

| 31  | 16 | 11 | 6 | 21 | 786 | /
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>RA</td>
<td>RB</td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

For each thread:
- If MAV = 1.0 then for TLB array = EA59:60
- If MAV = 2.0 then for each TLB entry
  - If MMUCFGTWC = 1 or TLBnCFGHES = 1 then
    - c = MAS6ISIZE
  - Else
    - c = entrySIZE
  - If (MV = 1.0) then
    - m = ¬((1 << (2×c)) - 1)
  - Else
    - m = ¬((1 << c) - 1)
  - If ((EA0:53 & m) = (entryEPN & m)) &
    - entrySIZE = MAS6ISIZE &
    - entryTID = MAS6SPID & entryTGS = MAS5SGS &
    - (E.PT not supported | entryTID = MAS5SIND) &
    - (E.HV not supported |
      - entryTLPID = MAS5SLPID &
      - entryTGS = MAS5SGS)) &
    - (MAV = 1.0) & (EA61 = 1)
    - then
    - if entryIPROT = 0 then entryV ← 0

Let the effective address (EA) be the sum (RA|0)+(RB). The EA is interpreted as shown below.

- **EA0:53**
  - **EA61:63** Reserved
- **EA59:60** TLB array selector [MAV = 1.0]
  - 00 TLB0
  - 01 TLB1
  - 10 TLB2
  - 11 TLB3
- **EA61** TLB invalidate all [MAV = 1.0]
- **EA62:63** Reserved

If EA61=0, all TLB entries on all threads that have all of the following properties are made invalid. The MAS registers listed are those in the thread executing the `tlbivax`.
- The MMU architecture version is 2.0 or the entry is in the TLB array targeted by EA59:60.
- The logical AND of EA0:53 and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is based on the following.
  - If MMUCFGTWC = 1 or TLBnCFGHES = 1, c is equal MAS6ISIZE. Otherwise, c is equal to entrySIZE.
  - If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 << (2×c)) - 1). Otherwise, m is equal to the logical NOT of ((1 << c) - 1).
- The TID value of the TLB entry is equal to MAS6SPID and the TS value of the TLB entry is equal to MAS6SAS.
- The implementation does not support the Embedded.Page Table category or the IND value of the TLB entry is equal to MAS5SIND.
- Either of the following is true:
  - The implementation does not support the Embedded.Hypervisor category.
  - The TLPIP value of the TLB entry is equal to MAS5SLPID and the TGS value of the TLB entry is equal to MAS5SGS.
  - entryIPROT = 0.

In MMU Architecture Version 1.0 if EA61=1, all entries in all threads not protected by the IPROT attribute in the TLB array targeted by EA59:60 are made invalid.

If the instruction specifies a TLB array that does not exist, the instruction is treated as if the instruction form is invalid. If the implementation requires the page size to be specified by MAS6ISIZE (MMUCFGTWC = 1 or, for the specified TLB array, TLBnCFGHES = 1) and the page size specified by MAS6ISIZE is not supported by the implementation, the instruction is treated as if the instruction form is invalid.

If the operation isn’t a TLB invalidate all and there are multiple entries in a single thread’s TLB array(s) that match the complete VPN, then zero or more matching entries with IPROT=0 are invalidated or a Machine Check interrupt occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

The operation performed by this instruction is ordered by the `mbar` (or `sync`) instruction with respect to a subsequent `tlbsync` instruction executed by the thread executing the `tlbivax` instruction. The operations caused by `tlbivax` and `tlbsync` are ordered by `mbar` as a set of operations which is independent of the other sets that `mbar` orders.

The effects of the invalidation on this thread are not guaranteed to be visible to the programming model.
until the completion of a context synchronizing operation.

Invalidations may occur for other TLB entries in the designated array, but in no case will any TLB entries with the IPROT attribute set be made invalid.

If RA does not equal 0, it is implementation-dependent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

**Special Registers Altered:**

None

---

**Programming Note**

Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

For backward compatibility, implementations may ignore a TLB entry's TS and TID fields when determining whether an entry should be invalidated. Since this and other such generous invalidation can be performed, consideration should be given to protecting a TLB entry that maps an interrupt vector by setting TLBIPROT = 1.

---

**Programming Note**

The `tlbilx` instruction is the preferred way of performing TLB invalidations for operating systems running as a guest to the hypervisor since the invalidations are partitioned and do not require hypervisor privilege.

---

**Programming Note**

The TLB invalidate all function (EA61=1) only exists in MMU Architecture Version 1.0 implementations. It should only be used when running existing software is deemed important.
**TLB Invalidate Local Indexed**

**X-form**

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>17</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>///</td>
<td>T</td>
<td>RA</td>
<td>RB</td>
<td>18</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)

for each TLB array
for each TLB entry
if MAV = 1.0 then
m ← ¬((1 << (2 × c)) - 1)
else
m ← ¬((1 << c) - 1)
if (entryIPROT = 0) & (entryTLPID = MAS5SLPID) then
if T = 0 then entryV ← 0
else
if T = 1 & entryTID = MAS6SPID then entryV ← 0
if T = 3 & entryTGS = MAS5SGS & ((EA0:53 & m) = (entryEPN & m)) & entrySIZE = MAS6ISIZE & entryTID = MAS6SPID & entryTS = MAS6SAS & (E.PT not supported | entryIND = MAS6SIND) then
entryV ← 0

Let the effective address (EA) be the sum (RA|0) + (RB).

The *tlbilx* instruction invalidates TLB entries in the thread that executes the *tlbilx* instruction. TLB entries which are protected by the IPROT attribute (entryIPROT = 1) are not invalidated.

If T = 0, all TLB entries that have all of the following properties are made invalid on the thread executing the *tlbilx* instruction.
- The TLPID of the entry matches MAS5SLPID.
- The IPROT of the entry is 0.

If T = 1, all TLB entries that have all of the following properties are made invalid on the thread executing the *tlbilx* instruction.
- The TLPID of the entry matches MAS5SLPID.
- The TID of the entry matches MAS6SPID.
- The IPROT of the entry is 0.

If T = 3, all TLB entries in the thread executing the *tlbilx* instruction that have all of the following properties are made invalid.
- The TLPID of the value of the TLB entry is equal to MAS5SLPID and the TGS value of the TLB entry is equal to MAS5SGS.
- The logical AND of EA0:53 and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is based on the following.

- If MMUCFGTWC = 1 or TLBnCFGHES = 1, c is equal MAS6ISIZE. Otherwise, c is equal to entrySIZE.
- If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 <<< (2 × c)) - 1). Otherwise, m is equal to the logical NOT of ((1 <<< c) - 1).
- The TID value of the TLB entry is equal to MAS6SPID and the TS value of the TLB entry is equal to MAS6SAS.
- The implementation does not support the Embedded.Page Table category or the IND value of the TLB entry is equal to MAS6SIND.
- The IPROT of entry is 0.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

Invalidations may occur for other TLB entries on the thread executing the *tlbilx* instruction, but in no case will any TLB entries with the IPROT attribute set be made invalid.

If T = 2, the instruction form is invalid.

If T = 3 and the implementation requires the page size to be specified by MAS6ISIZE (MMUCFGTWC = 1 or, for any TLB array, TLBnCFGHES = 1) and the page size specified by MAS6ISIZE is not supported by the implementation, the instruction is treated as if the instruction form is invalid.

If T=3 and there are multiple entries in the TLB array(s) that match the complete VPN, then zero or more matching entries with IPROT=0 are invalidated or a Machine Check interrupt occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

If RA does not equal 0, it is implementation-dependent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported and guest execution of TLB Management instructions is disabled (EPCR[DGTM]=1), this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

**Special Registers Altered:**

None

**Extended Mnemonics:**

Examples of extended mnemonics for *TLB Invalidate Local*:

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>tlbilxpid</td>
<td>tlbilx 0,0,0</td>
</tr>
<tr>
<td>tlbixpid</td>
<td>tlbilx 1,0,0</td>
</tr>
<tr>
<td>tlbilxva</td>
<td>tlbilx 3,RA,RB</td>
</tr>
<tr>
<td>tlbilxva</td>
<td>tlbilx 3,0,RB</td>
</tr>
</tbody>
</table>
Programming Note

tlbix is the preferred way of performing TLB invalidations, especially for operating systems running as a guest to the hypervisor since the invalidations are partitioned and do not require hypervisor privilege.

Programming Note

When dispatching a guest operating system, hypervisor software should always set MAS5SLPID to the guest's corresponding LPID value.

Programming Note

Executing a tlbix instruction with T=0 or T=1 may take many cycles to perform. Software should only issue these operations when an LPID or a PID value is reused or taken out of use.
TLB Search Indexed X-form

**tlbsx RA, RB**

| 0 | 31 | 6 | 11 | 16 | 21 | 914 | /

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)

Valid_matching_entry_exists ← 0

for each TLB array
  for each TLB entry
    if MAV = 1.0 then
      m ← ¬((1 << (2 × entrySIZE)) - 1)
    else
      m ← ¬((1 << entrySIZE) - 1)
    end if
    if ((EA0:53 & m) = (entryEPN & m)) &
      (entryTID = MAS6SPID | entryTID = 0) &
      (entryTS = MAS6SAS) &
      (E.PT not supported | entryIND = MAS6SIND) &
      (E.HV not supported | (entryTGS = MAS5SGS &
      (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
      Valid_matching_entry_exists ← 1
      exit for loops
    end if
  end for each TLB entry
  if Valid_matching_entry_exists
    entry ← matching entry found
    array ← TLB array number where TLB entry found
    index ← index into TLB array of TLB entry found
    if TLB array supports Next Victim then
      hint ← hardware hint for Next Victim
    else
      hint ← undefined
      rp ← entryYEM
      MAS0ATSEL ← 0
      MAS0TLBSEL ← array
      MAS0SEL ← index
      if MAV supported
        MAS0NV ← 0
      end if
      if Next Victim supported then
        if TLB array specified by MAS0TLBSEL supports NV
          then
          MAS0NV ← hint
          else
          MAS0NV ← undefined
        end if
        MAS1V ← 1
        MAS1TID TS TSIZE ← entryTID TS SIZE
        if TLB array supports IPROT then
          MAS1IPROT ← entryIPROT
        else
          MAS1IPROT ← 0
        end if
        if category E.PT supported then
          if TLB array supports indirect entries then
            MAS1IND ← entryIND
            if entryIND = 1
              MAS1TID ← entryTID
            else
              MAS1IND ← 0
            end if
            if (E.PT not supported | entryIND = 0) &
              (E.HV not supported | (entryTGS = MAS5SGS &
              (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
              Valid_matching_entry_exists ← 1
              exit for loops
            end if
          end if
        end if
        if category E.HV supported then
          MAS1TID ← entryTID
        else
          MAS1TID ← 0
        end if
        if (E.PT not supported | entryIND = 0) &
          (E.HV not supported | (entryTGS = MAS5SGS &
          (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
          Valid_matching_entry_exists ← 1
          exit for loops
        end if
        if category E.TWC supported then
          MAS1W ← 0b01
          if category E.PT supported then
            MAS1IND ← entryIND
            if entryIND = 1
              MAS1TID ← entryTID
            else
              MAS1IND ← 0
            end if
            if (E.PT not supported | entryIND = 0) &
              (E.HV not supported | (entryTGS = MAS5SGS &
              (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
              Valid_matching_entry_exists ← 1
              exit for loops
            end if
          end if
        end if
        if category E.HV supported then
          MAS1TID ← entryTID
        else
          MAS1TID ← 0
        end if
        if (E.PT not supported | entryIND = 0) &
          (E.HV not supported | (entryTGS = MAS5SGS &
          (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
          Valid_matching_entry_exists ← 1
          exit for loops
        end if
        if category E.TWC supported then
          MAS1W ← 0b01
          if category E.PT supported then
            MAS1IND ← entryIND
            if entryIND = 1
              MAS1TID ← entryTID
            else
              MAS1IND ← 0
            end if
            if (E.PT not supported | entryIND = 0) &
              (E.HV not supported | (entryTGS = MAS5SGS &
              (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
              Valid_matching_entry_exists ← 1
              exit for loops
            end if
          end if
        end if
        if category E.HV supported then
          MAS1TID ← entryTID
        else
          MAS1TID ← 0
        end if
        if (E.PT not supported | entryIND = 0) &
          (E.HV not supported | (entryTGS = MAS5SGS &
          (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
          Valid_matching_entry_exists ← 1
          exit for loops
        end if
        if category E.TWC supported then
          MAS1W ← 0b01
          if category E.PT supported then
            MAS1IND ← entryIND
            if entryIND = 1
              MAS1TID ← entryTID
            else
              MAS1IND ← 0
            end if
            if (E.PT not supported | entryIND = 0) &
              (E.HV not supported | (entryTGS = MAS5SGS &
              (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
              Valid_matching_entry_exists ← 1
              exit for loops
            end if
          end if
        end if
        if category E.HV supported then
          MAS1TID ← entryTID
        else
          MAS1TID ← 0
        end if
        if (E.PT not supported | entryIND = 0) &
          (E.HV not supported | (entryTGS = MAS5SGS &
          (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
          Valid_matching_entry_exists ← 1
          exit for loops
        end if
      end if
    end if
  end if
end for each TLB array
end for each TLB entry
end for each TLB array
end for each TLB entry
end for each TLB array
end for each TLB entry
end if

Let the effective address (EA) be the sum (RA|0)+ (RB).
If the search is successful, MAS register fields are loaded from the matching TLB entry according to the following:

- **MAS0ATSEL** is set to 0.
- **MAS0TLBSEL** is set to the number of the TLB array with the matching entry.
- **MAS0ESEL** is set to the index of the matching entry.
- If **MAS0HES** is supported, **MAS0HES** is set to 0.
- If Next Victim is supported for any TLB array, the following applies:
  - If the TLB array with the matching entry supports Next Victim, **MAS0NV** is set to 0.
  - Otherwise, **MAS0NV** is set to an implementation-dependent undefined value.
- **MAS1V** is set to 1.
- **MAS1TIDTS** are loaded from the TID, TS, and SIZE fields of the TLB entry.
- If the TLB array supports IPROT, **MAS1IPROT** is loaded from the IPROT bit of the TLB entry. Otherwise, **MAS1IPROT** is set to 0.
- **MAS2EPN** are loaded from the EPN, W, I, M, G, and E fields of the TLB entry.
- If the VLE category is supported, **MAS2VLE** is loaded from the VLE bit of the TLB entry.
- If Alternate Coherency Mode is supported, **MAS2ACM** is loaded from the ACM bit of the TLB entry.
- **MAS3RPNL** is loaded from the lower 22-bits of the RPN field of the TLB entry, and, if implemented, **MAS7RPNU** is loaded from the upper 32-bits of the RPN field of the TLB entry.
- The supported User-Defined storage control bits in **MAS3UX** are loaded from the respective supported U0:U3 bits of the TLB entry.
- If the Embedded.Page Table category is not supported, **MAS3UX** are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry. Otherwise, the following applies.
  - If the TLB array does not support indirect entries, **MAS1IND** is set to 0 and **MAS3UX** are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry. Otherwise, the following applies.
    - **MAS1IND** is loaded from the IND bit of the TLB entry.
    - If the IND bit of the TLB entry is 1, **MAS3SPSIZE** is loaded from the SPSIZE field of the TLB entry, and **MAS3UND** is set to an implementation-dependent undefined value.
    - If the IND bit of the TLB entry is 0, **MAS3UX** are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry.
- If the Embedded.Hypervisor category is implemented, **MAS8TGS** are loaded from the TGS, VF, and TLPID fields of the TLB entry.

If no valid matching translation exists, **MAS1V** is set to 0 and the MAS register fields are loaded according to the following in order to facilitate a TLB replacement:

- **MAS0ATSEL** is set to 0.
- **MAS0TLBSEL** is loaded from **MAS4TLBSELD**.
- If Next Victim is not supported for any TLB array, **MAS0ESEL** is set to an implementation-dependent undefined value. Otherwise, the following applies.
  - If the TLB array specified by **MAS4TLBSELD** supports Next Victim, **MAS0NV** is set to the hardware hint for the index of the entry to be replaced and **MAS0NV** is set to the hardware hint for the index of the next entry to be replaced. Otherwise, **MAS0ESEL** and **MAS0NV** are set to implementation-dependent undefined values.
- If **MAS0HES** is supported, **MAS0HES** is set to the value of **TLBnCFGHES** for the TLB array specified by **MAS4TLBSELD**.
- **MAS1IPROT** is set to 0.
- **MAS1TIDTS** are loaded from **MAS6SPID**.
- **MAS1TSIZE** is loaded from **MAS4TSIZED**.
- If the Embedded.Page Table category is supported, **MAS1IND** is set to **MAS4IND**.
- **MAS2EPN** is set to an implementation-dependent undefined value.
- **MAS2W** are loaded from **MAS4WD**.
- **MAS2ACM** is loaded from **MAS4ACMD**.
- **MAS2VLE** is loaded from **MAS4VLED**.
- **MAS2ACM** and, if implemented, **MAS7RPNU** are set to 0.
- The supported User-Defined storage control bits in **MAS3UX** are set to 0.
- **MAS3UX** are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry.
- **MAS0WQ** is set to 0b01.

If a **tlbsx** is successful, it is considered to “hit”. Otherwise, it is considered to “miss”.

If there are multiple matching TLB entries, either one of the matching entries is used or a Machine Check exception occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

If RA does not equal zero, it is implementation-dependent whether an illegal instruction exception occurs.

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

**Special Registers Altered:**
- **MAS0**
- **MAS1**
- **MAS2**
- **MAS3**
- **MAS7**
- **MAS8** (if category E.HV supported)
TLB Search and Reserve Indexed  X-form

tlbsrx. RA, RB [Category: Embedded.TLB Write Conditional]

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 |    |    | 1  |    | 2  |    | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
pid ← MAS1TID
as ← MAS1TS
if Embedded.Page Table category supported then
ind ← MAS1IND
if category E.HV supported then
gs ← MAS5SGS
lpid ← MAS5SLPID
va ← gs || lpid || as || pid || EA
else
va ← as || pid || EA
TLB-RESERVE ← 1
if Embedded.Page Table category supported then
TLB-RESERVE_ADDR ← ind || va
else
TLB-RESERVE_ADDR ← va
Valid_matching_entry_exists ← 0
for each TLB array
for each TLB entry
if MAV = 1.0 then
  m ← ¬((1 << (2 × entrySIZE)) - 1)
else
  m ← ¬((1 << entrySIZE) - 1)
if ((EA0:53 & m) = (entryEPN & m)) & (entryTID = MAS1TID | entryTID = 0) & entryTS = MAS1TS &
  (E.PT not supported | entryIND = MAS1IND) &
  (E.HV not supported | (entryTGS = MAS5SGS & entryTLPID = MAS5SLPID | entryTLPID = 0)) then
  Valid_matching_entry_exists ← 1
exit for loops
if Valid_matching_entry_exists then
  CR0 ← 0b0010
else
  CR0 ← 0b0000

Let the effective address (EA) be the sum (RA|0)+
(RB).

If any TLB array contains a valid entry matching the MAS1IND <E.PT> and virtual address formed by
MAS5SGS <E.HV>, MAS5SLPID <E.HV>, MAS1TS TID, and EA, the search is considered successful. A TLB
entry matches if all the following conditions are met.

- The valid bit of the TLB entry is 1.
- Either the Embedded.Page Table category is not supported or the IND value of the TLB entry is equal to
  MAS1IND.
- The logical AND of EA0:53 and m is equal to the logical AND of the EPN value of the TLB entry and m,
  where m is determined as follows:
  - If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 << entrySIZE) - 1).
  - Otherwise, m is equal to the logical NOT of ((1 << entrySIZE) - 1).
- The TID value of the TLB entry is equal to MAS1TID or is zero.
- The TS value of the TLB entry is equal to MAS1TS.
- Either of the following is true:
  - The implementation does not support the Embedded.Hypervisor category.
  - The TGS value of the TLB entry is equal to MAS5SGS and either the TLPID value of the TLB entry is equal to
    MAS5SLPID or is zero.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the search was successful.

CR0LT GT EQ SO = 0b00 || n || 0

This instruction creates a TLB-reservation for use by a TLB Write instruction. The virtual address described
above is associated with the TLB-reservation, and replaces any address previously associated with
the TLB-reservation. (The TLB-reservation is created regardless of whether the search succeeds.)

If there are multiple matching TLB entries, either one of
the matching entries is used or a Machine Check
exception occurs. If the Embedded.Hypervisor cate-
gory is supported, this Machine Check interrupt must
be precise.

If RA does not equal zero, it is implementation-depen-
dent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported and
guest execution of TLB Management instructions is dis-
abled (EPCR DGTMI=1), this instruction is hypervisor
privileged. Otherwise, this instruction is privileged.

Special Registers Altered:
CR0

1138  Power ISA™ - Book III-E
TLB Read Entry

if MAS0ATSEL = 0 then
  if TLBnCFGHES = 0 then
    entry ← SelectTLB(MAS0TLBSEL, MAS1TID TSIZE, MAS2EPN, MAS0ESEL)
  else
    entry ← SelectTLB(MAS0TLBSEL, MAS1TID TSIZE, MAS2ESEL, MAS0ESEL)
if Next Victim supported then
  if TLB array specified by MAS0TLBSEL supports NV then
    MAS0NV ← hint
  else
    MAS0NV ← undefined
if TLB entry is found then
  rpm ← entryYn
  MAS1V TID TSIZE ← entryYn TID TSIZE
  if TLB array supports IPROT then
    MAS1IPROT ← entryIPROT
  else
    MAS1IPROT ← 0
  if category E.PT supported then
    if TLB array supports indirect entries then
      if category E.PT supported then
        if entryIND = 1
          MAS1IND ← entryIND
        else
          MAS1IND ← 0
        end
        MAS3UX SX UN SW UR SR ← entryUX SX UN SW UR SR
      else
        MAS3UX SX UN SW UR SR ← entryUX SX UN SW UR SR
      end
    else
      MAS3UX SX UN SW UR SR ← entryUX SX UN SW UR SR
    end
else
  MAS3UX SX UN SW UR SR ← entryUX SX UN SW UR SR
end
  MAS2EPN W I M G E ← entryEPN W I M G E
end
if value VLE supported then
  if ACM supported then
    MAS2ACM ← entryACM
  else
    MAS2ACM ← undefined
  end
  if category VLE supported then
    if Embedded.Page Table supported then
      MAS2ESEL ← entryESEL
    else
      MAS2ESEL ← undefined
    end
else
  MAS2ESEL ← undefined
end
if Next Victim is supported for any TLB array, the following applies.
  If the Embedded.Hypervisor.LRAT category is not supported or MAS0ATSEL = 0, then the following applies.
  - If Next Victim is supported for any TLB array, the following applies.
    - If the TLB array specified by MAS0TLBSEL supports Next Victim, MAS0NV is set to the hardware hint for the index of the entry to be replaced. Otherwise, MAS0NV is set to a non-hint value.
  - A TLB entry is selected in one of the following ways.
    - If TLBnCFGHES = 0 for the TLB array selected by MAS0TLBSEL, the TLB entry is specified by MAS0TLBSEL, MAS0ESEL and MAS2EPN.
    - If TLBnCFGHES = 1 for the TLB array selected by MAS0TLBSEL, the TLB entry is specified by MAS0TLBSEL, MAS0ESEL and a hardware generated hash based on MAS1TID, MAS1TSIZE, and MAS2ESEL.
  - If the selected TLB entry exists, MAS register fields are loaded according to the following.
    - MAS1V TID TSIZE are loaded from the V, TID, TS, and SIZE fields of the TLB entry.
    - If the TLB array supports IPROT, MAS1IPROT is loaded from the IPROT bit of the TLB entry. Otherwise, MAS1IPROT is set to 0.
    - If ACM supported then MAS2ACM is set to 0.
    - If VLE category is supported, MAS2VLE is loaded from the VLE bit of the TLB entry.
    - If Alternate Coherency Mode is supported, MAS2ACM is loaded from the ACM bit of the TLB entry.
MAS3RPNL is loaded from the lower 22-bits of the RPN field of the TLB entry, and, if implemented, MAS7RPNU is loaded from the upper 32-bits of the RPN field of the TLB entry.

The supported User-Defined storage control bits in MAS3UX:U3 are loaded from the respective supported UX:U3 bits of the TLB entry.

If the Embedded.Page Table category is not supported, MAS3UX:UX, SW:SW, UR:UR, and SR:SR bits of the TLB entry. Otherwise, the following applies.

- If the TLB array does not support indirect entries, MAS1IND is set to 0 and MAS3UX:UX, UW:UW, SW:SW, UR:UR, and SR:SR bits of the TLB entry. Otherwise, the following applies.
  - If the IND bit of the TLB entry is 1, MAS1V is set to 0 and the following MAS register fields are set to implementation-dependent undefined values.
  - If the IND bit of the TLB entry is 0, MAS3UX:UX, SW:SW, UR:UR, and SR:SR bits of the TLB entry.

If the Embedded.Hypervisor category is implemented, MAS8TGS:VFTLPID are loaded from the TGS, VF, and TLPID fields of the TLB entry.

If the Embedded.Hypervisor category is supported, the LRAT array is specified (MAS0ATSEL = 1), then the following applies.

- If the LRAT entry specified by MAS0ESEL and MAS2EPN exists, MAS register fields are loaded from the LRAT entry according to the following.
  - MAS1V is loaded from the V and LSIZE fields of the LRAT entry.
  - MAS1IPROT TID:TSIZE, MAS2EPN W I M G E, and MAS3UX:UX, SW:SW, UR:UR, and SR:SR are set to 0s.
  - If the Embedded.Page Table category is supported, MAS1IND is set to 0.
  - If the VLE category is supported, MAS2VLE is set to 0.
  - If Alternate Coherency Mode is supported, MAS2ACM is set to 0.
  - If Alternate Coherency Mode is supported, MAS3RPNL is loaded from the lower 22-bits of the LRPN field of the LRAT entry, and, if implemented, MAS7RPNU is loaded from the upper 32-bits of the LRPN field of the LRAT entry.
  - The supported User-Defined storage control bits in MAS3UX:UX are set to 0s.
  - MAS8TGS:VF are set to 0s.

If the Embedded.Page Table category is supported (TLBnCFGoured = 1), MAS8TLPID is loaded from the TLPID field of the LRAT entry.

If TLBnCFGoured = 1 and the page size specified by MAS1TSIZE is not supported by the specified array, the *tlbre* may be performed as if TSIZE were some implementation-dependent value of or, as described below, as if the entry can not be found, or an Illegal Instruction exception occurs.

It is implementation-dependent whether a TLB or LRAT entry can not be found or whether larger values of the fields that select an entry are simply mapped to existing entries. If the specified TLB or LRAT entry does not exist, MAS1V is set to 0 and the following MAS register fields are set to implementation-dependent undefined values.

- MAS1IPROT TID:TSIZE, MAS2EPN W I M G E, and MAS3UX:UX, SW:SW, UR:UR, and SR:SR are set to 0s.
- If the Embedded.Page Table category is supported, MAS1IND is set to 0.
- If the VLE category is supported, MAS2VLE is set to 0.
- If Alternate Coherency Mode is supported, MAS2ACM is set to 0.
- The supported User-Defined storage control bits in MAS3UX:UX are set to 0s.
- If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

**Special Registers Altered:**

- MAS0
- MAS1
- MAS2
- MAS3
- MAS8 (if category E.HV is supported)

**Programming Note**

Hypervisor software should generally prevent guest operating system visibility of the RPN. After executing a *tlbsx* or *tlbre* on behalf of a guest, the hypervisor should replace the RPN fields in the MAS3 and MAS7 registers with the corresponding values from the appropriate LPN.
Chapter 6. Storage Control

**TLB Synchronize X-form**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>566</th>
<th>31</th>
</tr>
</thead>
</table>

The `tlbsync` instruction provides an ordering function for the effects of all `tlbivax` instructions executed by the thread executing the `tlbsync` instruction, with respect to the memory barrier created by a subsequent `sync` instruction executed by the same thread. Executing a `tlbsync` instruction ensures that all of the following will occur.

- All TLB invalidations caused by `tlbivax` instructions preceding the `tlbsync` instruction will have completed on any other thread before any data accesses caused by instructions following the `sync` instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated will have been performed with respect to the thread executing the `sync` instruction, to the extent required by the associated Memory Coherence Required attributes, before the `sync` instruction's memory barrier is created.

The operation performed by this instruction is ordered by the `mbar` or `sync` instruction with respect to preceding `tlbivax` instructions executed by the thread executing the `tlbsync` instruction. The operations caused by `tlbivax` and `tlbsync` are ordered by `mbar` as a set of operations, which is independent of the other sets that `mbar` orders.

The `tlbsync` instruction may complete before operations caused by `tlbivax` instructions preceding the `tlbsync` instruction have been performed.

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

**Special Registers Altered:**

- None

---

**Programming Note**

Care must be taken on some implementations when using the `tlbsync` instruction as there may be a system-imposed restriction of only one `tlbsync` allowed on the bus at a given time in the system.

**TLB Write Entry X-form**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>4</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>978</th>
<th>31</th>
</tr>
</thead>
</table>

if `MAS0WQ = 0b00 | MAS0HES = 0b01` then
if `MAS0ATSEL = 0` or `MSRGS = 1` then
entry `← SelectTLB(MAS0TLBSEL, MAS0ESEL, MAS2EPN)`
else
  if `MAS0HES = 1` then
    entry `← SelectTLB(MAS0TLBSEL, MAS1TID, TSIZE, MAS2EPN, hardware_replacement_algorithm)`
  else
    entry `← SelectTLB(MAS0TLBSEL, MAS1TID, TSIZE, MAS2EPN, MAS0ESEL)`
if TLB array specified by `MAS0TLBSEL` supports NV & (`MAS0WQ = 0b00` | (category E.TWC supported & (`MAS0WQ = 0b01` & (TLB-reservation))) then
  hint `← MAS0NV`
if TLB entry is found & (`MAS0WQ = 0b00` | ((category E.TWC supported) & (`MAS0WQ = 0b01` & (TLB-reservation))) then
if category E.HV.LRAR supported & (MSRGS=1) & (`MAS1ATSEL = 0`) then
  rpn `← translate_logical_to_real(MAS7RPNU || MAS3RPNL, MAS3U0:U3)`
else
  if `MAS7 implemented` then
    rpn `← MAS7RPNU || MAS3RPNL`
  else
    rpn `← 320 || MAS3RPNL`
entryV LSIZE
if category E.PT supported and TLB array supports indirect entries then
entryIND `← MAS1IND`
if `MAS1IND = 0` then
  entryUX `← MAS3UX || MAS3U0:U3`
else
  entrySPSIZE `← MAS3RPLS`
  entryUX `← MAS3UX || MAS3U0:U3`
entryRPN `← rpn`
if (category E.HV is supported) then
entryTGS VF TLPID `← MAS8TGS VF TLPID`
if category E.TWC supported
TLB-reservation `← 0`
else
  entry `← SelectLRAT(MAS0ESEL, MAS2EPN)`
  if LRAT entry is found & (`MAS0WQ = 0b00`) & (MAS0HES = 0b0) then
    hint `← MAS0NV`
  entryRLR `← pull(MAS1VPNL, hardware_replacement_algorithm)`
  else
    if category E.TWC supported
      TLB-reservation `← 0`
If the Embedded.TLB Write Conditional category is not supported, MAS0WQ is treated as if it were 0b00 in the following description.

If the Embedded.Hypervisor.LRAT category is not supported or MSRGS=1, MAS0ATSEL is treated as if it were zero in the following description.

If a TLB array is specified (MAS0ATSEL = 0 or MSRGS = 1) and TLBnCFGHES=0 for the TLB array selected by MAS0TLBSEL, MAS0HES is treated as 0 in the following description.

If the Embedded.Page Table category is supported, a TLB array is specified (MAS0ATSEL = 0 or MSRGS = 1), and the specified TLB array does not support indirect entries, MAS1IND is treated as 0.

If a TLB array is specified (MAS0ATSEL = 0 or MSRGS = 1) and MAS0WQ is 0b00 or 0b01, the following applies.

- If the Embedded.Hypervisor is not supported, the tlbwe instruction is executed in hypervisor state, or MAS1v=0, an RPN is formed by concatenating MAS7RPNU with MAS3RPNL (RPN = MAS7RPNU || MAS3RPNL).
- If the Embedded.Hypervisor category is supported, the tlbwe instruction is executed in guest state, and MAS1v=1, an LPN is formed by concatenating MAS7RPNU with MAS3RPNL (LPN = MAS7RPNU || MAS3RPNL). However, if MAS7 is not implemented, LPN = 320 || MAS3RPNL. This LPN is translated by the LRAT to obtain the RPN. If there is no LRAT entry that translates this LPN for the LPIID specified by MAS8TLPID an LRAT Miss exception occurs. However, if MAS0WQ is 0b01 and no TLB-reservation exists, it is implementation-dependent whether the LRAT Miss exception occurs.
- If TLBnCFGHES for the TLB array selected by MAS0TLBSEL is 0, the TLB entry is specified by MAS0TLBSEL, MAS0SESEL, and MAS2EPN. If TLBnCFGHES is 1 and MAS0HES is 1, the TLB entry is selected by MAS0TLBSEL, a hardware replacement algorithm, and a hardware generated hash based on MAS1TID TSIZE, and MAS2EPN. If TLBnCFGHES is 1 and MAS0HES is 0, the TLB entry is selected by MAS0TLBSEL, MAS0SESEL and a hardware generated hash based on MAS0TLBSEL, MAS1TID TSIZE, and MAS2EPN.
- The selected TLB entry is written (see the following major bulleted item) if all the following conditions are met.
  - There is no LRAT Miss exception.
  - MAS0WQ is 0b00 or both the following are true.
    - MAS0WQ is 0b01
    - A TLB-reservation exists.
  - MAS1IPROT is 0, the Embedded.Hypervisor category is not supported, or MSRGS=0.
  - The selected TLB entry has IPROT = 0, the Embedded.Hypervisor category is not supported, or MSRGS=0.
- If the Embedded.Hypervisor category is supported, use the first of the following sub-bullets that applies.
  - If EPCR<DTMI>=1 and MSRGS=1, no TLB entry is written and a Hypervisor Privilege exception occurs.
  - If the selected entry exists, the selected entry has IPROT=1, MAS0WQ=0b00, and MSRGS=1, no TLB entry is written, and a Hypervisor Privilege exception occurs.
  - If MAS0WQ=0b00, MAS1v=1, MAS1IPROT=1, and MSRGS=1, no TLB entry is written, and it is implementation-dependent whether a Hypervisor Privilege exception occurs.
  - If MAS0WQ=0b00, MAS1v=0, MAS1IPROT=1, and MSRGS=1, no TLB entry is written, and it is implementation-dependent whether a Hypervisor Privilege exception occurs.

If a TLB entry is to be written per the preceding description, then regardless of whether the selected TLB entry exists, MAS0VU provides a suggestion to hardware of what the hardware hint for replacement should be when the next Data or Instruction TLB Error Interrupt for a virtual address that uses the set of TLB entries containing the entry written by the tlbwe instruction.

If the selected TLB entry exists and the TLB entry is to be written per the preceding description, the fields of the TLB entry are loaded from the MAS registers according to the following.

- The V, TID, TS, and SIZE fields of the TLB entry are loaded from MAS1V TID TS SIZE.
- If the TLB array supports IPROT, the IPROT bit of the TLB entry is loaded from MAS1IPROT.
- The EPN, W, I, M, G, and E fields of the TLB entry are loaded from MAS2EPN W I M G E.
- If the VLE category is supported, the VLE bit of the TLB entry is loaded from MAS2VLE.
- If Alternate Coherency Mode is supported, the ACM bit of the TLB entry is loaded from MAS2ACM.
- The RPN field of the TLB entry is loaded from the RPN described above.
- The supported User-Defined storage control bits (U0:U3) of the TLB entry are loaded from the respective bits in MAS3U0:U3.
- If the Embedded.Page Table category is supported and the TLB array supports indirect entries, the following applies.
The IND of the TLB entry is loaded from MAS1\textsubscript{IND}.
- If MAS1\textsubscript{IND} is 1, the SPSIZE field of the TLB entry is loaded from MAS3\textsubscript{SPSIZE}.
- If MAS1\textsubscript{IND} is 0, the UX, SX, UW, SW, UR, and SR bits of the TLB entry are loaded from MAS3\textsubscript{UX} SX UW SW UR SR.
- If the Embedded.Page Table category is not supported or the TLB array does not support indirect entries, the UX, SX, UW, SW, UR, and SR bits of the TLB entry are loaded from MAS3\textsubscript{UX} SX UW SW UR SR.
- If the Embedded.Hypervisor category is implemented, the TGS, VF, and TLPID fields of the TLB entry are loaded from MAS8\textsubscript{TGS} VF TLPID.

If the LRAT array is specified (MAS0\textsubscript{ATSEL} = 0 or MSR\textsubscript{GS} = 1) for a \texttt{tlbwe}, MAS0\textsubscript{WQ} must be 0b00 and MAS0\textsubscript{HES} must be 0. If the LRAT array is specified (MAS0\textsubscript{ATSEL} = 0 or MSR\textsubscript{GS} = 1), MAS0\textsubscript{WQ} is 0b00, MAS0\textsubscript{HES} is 0, and the \texttt{tlbwe} instruction is executed in hypervisor state, the following applies.
- An RPN is formed by concatenating MAS7\textsubscript{RPNU} with MAS3\textsubscript{RPNL} (RPN = MAS7\textsubscript{RPNU} || MAS3\textsubscript{RPNL}).
- The contents of the MAS1\textsubscript{V} TSIZE, MAS2\textsubscript{EPN}, and the RPN described above are written to the selected LRAT entry\textsubscript{V} LSIZE LPN RPN.
- If the LPID field in the LRAT is supported (LRATCFG\textsubscript{LPID} = 1), MAS8\textsubscript{TLPID} is written to the LPID field of the selected entry.

If no exception occurs, and either MAS0\textsubscript{WQ} is 0b10 or a TLB array was selected by the \texttt{tlbwe} (MAS0\textsubscript{ATSEL}=0 or MSR\textsubscript{GS}=1), the TLB-reservation is cleared.
- If MAS0\textsubscript{WQ} is 0b10, no TLB entry is written.
- If MAS0\textsubscript{WQ} is 0b11, the instruction is treated as if the instruction form is invalid.

If the page size specified by MAS1\textsubscript{TSIZE} is not supported by the specified array, the \texttt{tlbwe} may be performed as if TSIZE were some implementation-dependent value, or an Illegal Instruction exception occurs.

If a TLB entry is to be written per the preceding description, MAS1\textsubscript{IND}=1, and values of I,M,G, and E to be written to the TLB entry are inconsistent with storage that is not Caching Inhibited, Memory Coherence Required, not Guarded, and Big-Endian, the \texttt{tlbwe} may be performed as described or an Illegal Instruction exception occurs. Also, if a TLB entry is to be written per the preceding description, MAS1\textsubscript{IND}=1, and values of ACM and U0:U3 to be written to the TLB entry are inconsistent with the requirements that an implementation has for storage control attributes for a Page Table, the \texttt{tlbwe} may be performed as described or an Illegal Instruction exception occurs.

If an invalid value is specified for MAS0\textsubscript{TLBSEL} MAS0\textsubscript{ESEL} or MAS2\textsubscript{EPN}, either no TLB entry is written by the \texttt{tlbwe}, or the \texttt{tlbwe} is performed as if some implementation-dependent, valid value were substituted for the invalid value, or an Illegal Instruction exception occurs.

A context synchronizing instruction is required after a \texttt{tlbwe} instruction to ensure any subsequent instructions that will use the updated TLB or LRAT values execute in the new context.

If TLBnCFG\textsubscript{HES}=1 for the selected TLB array, a TLB write does not necessarily invalidate implementation-specific TLB lookaside information. See Section 6.11.4.4.

This instruction is hypervisor privileged if the Embedded.Hypervisor category is supported and any of the following is true.
- The Embedded.Hypervisor.LRAT category is not supported.
- MSR\textsubscript{GS} = 1 and, for the TLB array selected by MAS0\textsubscript{TLBSEL}, TLBnCFG\textsubscript{GTWE} = 0.
- Guest execution of \textit{TLB Management} instructions is disabled (EPCR\textsubscript{DGTMI}=1).

Otherwise, this instruction is privileged.

**Special Registers Altered:**
None

### Programming Note
Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.
Chapter 7. Interrupts and Exceptions

7.1 Overview

An interrupt is the action in which the thread saves its old context (MSR and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. Exceptions are the events that will, if enabled, cause the thread to take an interrupt.

Exceptions are generated by signals from internal and external peripherals, instructions, the internal timer. Interrupts are divided into 4 classes, as described in Section 7.4.3, such that only one interrupt of each class is reported, and when it is processed no program state is lost. Since Save/Restore register pairs GSRR0/GSRR1 <E.HV> SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: E.ED], and MCSSR0/MCSSR1 are serially reusable resources used by guest <E.HV>, base, critical, debug [Category: E.ED], Machine Check interrupts, respectively, program state may be lost when an unordered interrupt is taken. (See Section 7.8, "Interrupt Ordering and Masking".

7.2 Interrupt Registers

7.2.1 Save/Restore Register 0

Save/Restore Register 0 (SRR0) is a 64-bit register. SRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an rfi is executed. On a non-critical interrupt, SRR0 is set to the current or next instruction address. When rfi is executed, instruction execution continues at the address in SRR0.

```
<table>
<thead>
<tr>
<th>SRR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Figure 53. Save/Restore Register 0

In general, SRR0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced.

The contents of SRR0 when an interrupt is taken are mode dependent, reflecting the computation mode when the interrupt is taken and the computation mode entered for execution of the interrupt (specified by EPCRiCm) <E.HV>. When computation mode when the interrupt is taken is 32-bit mode and the computation mode entered for execution of the interrupt is 64-bit mode, the high-order 32 bits of SRR0 are set to 0s. When computation mode when the interrupt is taken is 64-bit mode and the computation mode entered for execution of the interrupt is 32-bit mode, the contents SRR0 are undefined.

The contents of SRR0 upon interrupt can be described as follows (assuming Addr is the address to be put into SRR0):

- if (MSRCM = 0) & (EPCRICM = 0) then SRR0 ← 32undefined || Addr32:63
- if (MSRCM = 0) & (EPCRICM = 1) then SRR0 ← 320 || Addr32:63
- if (MSRCM = 1) & (EPCRICM = 1) then SRR0 ← Addr0:63
- if (MSRCM = 1) & (EPCRICM = 0) then SRR0 ← undefined

The contents of SRR0 can be read into register RT using mfspr RT,SRR0. The contents of register RS can be written into the SRR0 using mtspr SRR0,RS.

This register is hypervisor privileged.

7.2.2 Save/Restore Register 1

Save/Restore Register 1 (SRR1) is a 32-bit register. SRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an rfi is executed. On a non-critical interrupt, SRR0 is set to the current or next instruction address. When rfi is executed, instruction execution continues at the address in SRR0.

```
<table>
<thead>
<tr>
<th>SRR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Figure 54. Save/Restore Register 1
Bits of SRR1 that correspond to reserved bits in the MSR are also reserved.

**Programming Note**

A MSR bit that is reserved may be inadvertently modified by \texttt{rfi/rfci/rfci}.  

The contents of SRR1 can be read into register RT using \texttt{mfspr RT,SRR1}. The contents of register RS can be written into the SRR1 using \texttt{mtspr SRR1,RS}.

This register is hypervisor privileged.

### 7.2.3 Guest Save/Restore Register 0 [Category:Embedded.Hypervisor]

Guest Save/Restore Register 0 (GSRR0) is a 64-bit register. GSRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on guest interrupts, and to restore machine state when an \texttt{rfi} is executed. On a guest interrupt, GSRR0 is set to the current or next instruction address. When \texttt{rfi} is executed, instruction execution continues at the address in GSRR0.

**Figure 55. Guest Save/Restore Register 0**

In general, GSRR0 contains the address of the instruction that caused the guest interrupt, or the address of the instruction to return to after a guest interrupt is serviced.

The contents of GSRR0 when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR\textsubscript{CM}) and the computation mode entered for execution of the interrupt (specified by EPCR\textsubscript{GCM}). The contents of GSRR0 upon interrupt can be described as follows (assuming Addr is the address to be put into GSRR0):

\[
\begin{align*}
\text{if (MSR}_{\text{CM}} = 0) \& (\text{EPCR}_{\text{GCM}} = 0) & \text{ then GSRR0 } \leftarrow \text{undefined } || \text{Addr}_{32:63} \\
\text{if (MSR}_{\text{CM}} = 0) \& (\text{EPCR}_{\text{GCM}} = 1) & \text{ then GSRR0 } \leftarrow 0 || \text{Addr}_{32:63} \\
\text{if (MSR}_{\text{CM}} = 1) \& (\text{EPCR}_{\text{GCM}} = 1) & \text{ then GSRR0 } \leftarrow \text{Addr}_{0:63} \\
\text{if (MSR}_{\text{CM}} = 1) \& (\text{EPCR}_{\text{GCM}} = 0) & \text{ then GSRR0 } \leftarrow \text{undefined}
\end{align*}
\]

The contents of GSRR0 can be read into register RT using \texttt{mfspr RT,GSRR0}. The contents of register RS can be written into the GSRR0 using \texttt{mtspr GSRR0,RS}.

This register is privileged.

### 7.2.4 Guest Save/Restore Register 1 [Category:Embedded.Hypervisor]

Guest Save/Restore Register 1 (GSRR1) is a 32-bit register. GSRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on guest interrupts, and to restore machine state when an \texttt{rfgi} is executed. When a guest interrupt is taken, the contents of the MSR are placed into GSRR1. When \texttt{rfgi} is executed, the contents of GSRR1 are placed into the MSR.

**Figure 56. Guest Save/Restore Register 1**

Bits of GSRR1 that correspond to reserved bits in the MSR are also reserved.

**Programming Note**

A MSR bit that is reserved may be inadvertently modified by \texttt{rfi/rfgi/rfcf/rfci/rfmcf}.

The contents of GSRR1 can be read into register RT using \texttt{mfspr RT,GSRR1}. The contents of register RS can be written into the GSRR1 using \texttt{mtspr GSRR1,RS}.

This register is privileged.

### 7.2.5 Critical Save/Restore Register 0

Critical Save/Restore Register 0 (CSRR0) is a 64-bit register. CSRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an \texttt{rfci} is executed. When a critical interrupt is taken, the CSRR0 is set to the current or next instruction address. When \texttt{rfci} is executed,
instruction execution continues at the address in CSRR0.

Figure 57. Critical Save/Restore Register 0

In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.

The contents of CSRR0 when an interrupt is taken are mode dependent, reflecting the computation mode when the interrupt is taken and the computation mode entered for execution of the interrupt (specified by EPCR[ICM]) [Category:Embedded.Hypervisor]. If computation mode when the interrupt is taken is 32-bit mode and the computation mode entered for execution of the interrupt is 64-bit mode, the high-order 32 bits of CSRR0 are set to 0s. When computation mode when the interrupt is taken is 64-bit mode and the computation mode entered for execution of the interrupt is 32-bit mode, the contents CSRR0 are undefined.

The contents of CSRR0 upon critical interrupt can be described as follows (assuming Addr is the address to be put into CSRR0):

\[
\begin{align*}
\text{if } (\text{MSRCM} = 0) & \& (\text{EPCRICM} = 0) \text{ then } \text{CSRR0} \leftarrow 32\text{undefined} \parallel \text{Addr}_{32:63} \\
\text{if } (\text{MSRCM} = 0) & \& (\text{EPCRICM} = 1) \text{ then } \text{CSRR0} \leftarrow 320 \parallel \text{Addr}_{32:63} \\
\text{if } (\text{MSRCM} = 1) & \& (\text{EPCRICM} = 1) \text{ then } \text{CSRR0} \leftarrow \text{Addr}_{0:63} \\
\text{if } (\text{MSRCM} = 1) & \& (\text{EPCRICM} = 0) \text{ then } \text{CSRR0} \leftarrow \text{undefined}
\end{align*}
\]

The contents of CSRR0 can be read into register RT using \texttt{mfspr RT,CSRR0}. The contents of register RS can be written into CSRR0 using \texttt{mtspr CSRR0,RS}.

This register is hypervisor privileged.

7.2.6 Critical Save/Restore Register 1

Critical Save/Restore Register 1 (CSRR1) is a 32-bit register. CSRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an \texttt{rfci} is executed. When a critical interrupt is taken, the contents of the MSR are placed into CSRR1. When \texttt{rfci} is executed, the contents of CSRR1 are placed into the MSR.

Figure 58. Critical Save/Restore Register 1

Bits of CSRR1 that correspond to reserved bits in the MSR are also reserved.

\[
\begin{align*}
\text{CSRR0} & \quad \parallel \\
0 & \quad 62 \ 63
\end{align*}
\]

\textbf{Programming Note}

A MSR bit that is reserved may be inadvertently modified by \texttt{rfi/rfci/rfmci}.

The contents of CSRR1 can be read into bits 32:63 of register RT using \texttt{mfspr RT,CSRR1}, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the CSRR1 using \texttt{mtspr CSRR1,RS}.

This register is hypervisor privileged.

7.2.7 Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug]

Debug Save/Restore Register 0 (DSRR0) is a 64-bit register used to save machine state on Debug interrupts, and to restore machine state when an \texttt{rdi} is executed. When a Debug interrupt is taken, the DSRR0 is set to the current or next instruction address. When \texttt{rdi} is executed, instruction execution continues at the address in DSRR0.

\[
\begin{align*}
\text{DSRR0} & \quad \parallel \\
0 & \quad 62 \ 63
\end{align*}
\]

Figure 59. Debug Save/Restore Register 0

In general, DSRR0 contains the address of an instruction that was executing or just finished execution when the Debug exception occurred.

The contents of DSRR0 when an interrupt is taken are mode dependent, reflecting the computation mode when the interrupt is taken and the computation mode entered for execution of the interrupt (specified by EPCR[ICM]) [Category:Embedded.Hypervisor]. If computation mode when the interrupt is taken is 32-bit mode and the computation mode entered for execution of the interrupt is 64-bit mode, the high-order 32 bits of DSRR0 are set to 0s. When computation mode when the interrupt is taken is 64-bit mode and the computation mode entered for execution of the interrupt is 32-bit mode, the contents DSRR0 are undefined.

The contents of DSRR0 upon Debug interrupt can be described as follows (assuming Addr is the address to be put into DSRR0):

\[
\begin{align*}
\text{if } (\text{MSRCM} = 0) & \& (\text{EPCRICM} = 0) \text{ then } \text{DSRR0} \leftarrow 32\text{undefined} \parallel \text{Addr}_{32:63} \\
\text{if } (\text{MSRCM} = 0) & \& (\text{EPCRICM} = 1) \text{ then } \text{DSRR0} \leftarrow 320 \parallel \text{Addr}_{32:63} \\
\text{if } (\text{MSRCM} = 1) & \& (\text{EPCRICM} = 1) \text{ then } \text{DSRR0} \leftarrow \text{Addr}_{0:63} \\
\text{if } (\text{MSRCM} = 1) & \& (\text{EPCRICM} = 0) \text{ then } \text{DSRR0} \leftarrow \text{undefined}
\end{align*}
\]

The contents of DSRR0 can be read into register RT using \texttt{mfspr RT,DSRR0}. The contents of register RS can be written into DSRR0 using \texttt{mtspr DSRR0,RS}.

This register is hypervisor privileged.
7.2.8 Debug Save/Restore Register 1 [Category: Embedded.Enhanced Debug]

Debug Save/Restore Register 1 (DSRR1) is a 32-bit register used to save machine state on Debug interrupts, and to restore machine state when an rfdi is executed. When a Debug interrupt is taken, the contents of the Machine State Register are placed into DSRR1. When rfdi is executed, the contents of DSRR1 are placed into the Machine State Register.

![Figure 60. Debug Save/Restore Register 1](image)

Bits of DSRR1 that correspond to reserved bits in the Machine State Register are also reserved.

The contents of DSRR1 can be read into bits 32:63 of register RT using mfspr RT,DSRR1, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the DSRR1 using mtspr DSRR1,RS.

This register is hypervisor privileged.

7.2.9 Data Exception Address Register

The Data Exception Address Register (DEAR) is a 64-bit register. DEAR bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The DEAR contains the address that was referenced by a Load, Store or Cache Management instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt if either the Embedded.Hypervisor category is not supported or the interrupt is directed to the hypervisor state. The DEAR is identical in form and function to DEAR.

The contents of the DEAR when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_CM) and the computation mode entered for execution of the interrupt (specified by EPCR_QICM). The contents of the DEAR upon interrupt can be described as follows (assuming Addr is the address to be put into GDEAR):

if (MSR_CM = 0) & (EPCR_QICM = 0)
then DEAR ← 32:undefined || Addr32:63
if (MSR_CM = 0) & (EPCR_QICM = 1)
then DEAR ← 32:0 || Addr32:63
if (MSR_CM = 1) & (EPCR_QICM = 1)
then DEAR ← Addr0:63
if (MSR_CM = 1)&(EPCR_QICM = 0)
then DEAR ← undefined

The contents of DEAR can be read into register RT using mfspr RT,DEAR. The contents of register RS can be written into the DEAR using mtspr DEAR,RS.

This register is privileged.

--- Programming Note ---

**mfspr RT,DEAR** should be used to read GDEAR in guest supervisor state. **mtspr DEAR,RS** should be used to write GDEAR in guest supervisor state. See Section 2.2.1, "Register Mapping".

7.2.10 Guest Data Exception Address Register [Category: Embedded.Hypervisor]

The Guest Data Exception Address Register (GDEAR) is a 64-bit register. GDEAR bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The GDEAR contains the address that was referenced by a Load, Store or Cache Management instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt that was directed to the guest supervisor state. The GDEAR is identical in form and function to DEAR.

The contents of the GDEAR when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_CM) and the computation mode entered for execution of the interrupt (specified by EPCR_QICM). The contents of the GDEAR upon interrupt can be described as follows (assuming Addr is the address to be put into GDEAR):

if (MSR_CM = 0) & (EPCR_QICM = 0)
then GDEAR ← 32:undefined || Addr32:63
if (MSR_CM = 0) & (EPCR_QICM = 1)
then GDEAR ← 32:0 || Addr32:63
if (MSR_CM = 1) & (EPCR_QICM = 1)
then GDEAR ← Addr0:63
if (MSR_CM = 1)&(EPCR_QICM = 0)
then GDEAR ← undefined

The contents of GDEAR can be read into register RT using mfspr RT,GDEAR. The contents of register RS can be written into the GDEAR using mtspr GDEAR,RS.

This register is privileged.

7.2.11 Interrupt Vector Prefix Register

The Interrupt Vector Prefix Register (IVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The IVPR is used for Machine Check interrupt if the MCIVPR is not supported. The IVPR is used for other interrupts if Category E.HV is not supported or if the interrupt is directed to the hypervisor state. For these interrupts, the IVPR is used in one of the following ways.

- If Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported, the following applies. Bits 48:63 are reserved. Bits 0:47 of...
the Interrupt Vector Prefix Register provide the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets from the appropriate IVOR (provided in Section 7.6.1, "Interrupt Fixed Offsets [Category: Embedded.Phased-In]") are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, the following applies. IVPR[52:63] are reserved. Bits 0:51 of the Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the exception processing routines. The 12-bit exception vector offsets (provided in Section 7.6.1, "Interrupt Fixed Offsets [Category: Embedded.Phased-In]") are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

The contents of Interrupt Vector Prefix Register can be read into register RT using mfspr RT,IVPR. The contents of register RS can be written into Interrupt Vector Prefix Register using mtspr IVPR,RS. This register is hypervisor privileged.

7.2.12 Guest Interrupt Vector Prefix Register [Category: Embedded.Hypervisor]

The Guest Interrupt Vector Prefix Register (GIVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

If Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported, the following applies. GIVPR[48:63] are reserved. For interrupts directed to guest state, bits 0:47 of the Guest Interrupt Vector Prefix Register provides the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets (provided in Section 7.6.1, "Interrupt Fixed Offsets [Category: Embedded.Phased-In]") are concatenated to the right of bits 0:47 of the Guest Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, the following applies. GIVPR[52:63] are reserved. For interrupts directed to guest state, bits 0:51 of the Guest Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the exception processing routines. The 12-bit exception vector offsets (provided in Section 7.6.1, "Interrupt Fixed Offsets [Category: Embedded.Phased-In]") are concatenated to the right of bits 0:47 of the Guest Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

The contents of Guest Interrupt Vector Prefix Register can be read into register RT using mfspr RT,GIVPR. The contents of register RS can be written into Interrupt Vector Prefix Register using mtspr GIVPR,RS. Write access to this register is hypervisor privileged. Read access to this register is privileged.

Programming Note

mfspr RT,IVPR should be used to read GIVPR in guest supervisor state. mtspr IVPR,RS should be used to write GIVPR in guest supervisor state. Hypervisor software should emulate the accesses for the guest.
### 7.2.13 Exception Syndrome Register

The Exception Syndrome Register (ESR) is a 32-bit register. ESR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The ESR provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupts, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other ESR bits are cleared. Other interrupt types do not affect the contents of the ESR. The ESR does not need to be cleared by software. Figure 61 shows the bit definitions for the ESR.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Meaning</th>
<th>Associated Interrupt Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:35</td>
<td>Implementation-dependent</td>
<td>(Implementation-dependent)</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>PIL</td>
<td>Illegal instruction exception</td>
<td>Program</td>
</tr>
<tr>
<td>37</td>
<td>PPR</td>
<td>Privileged Instruction exception</td>
<td>Program</td>
</tr>
<tr>
<td>38</td>
<td>PTR</td>
<td>Trap exception</td>
<td>Program</td>
</tr>
<tr>
<td>39</td>
<td>FP</td>
<td>Floating-point operation</td>
<td>Alignment, Data Storage, Data TLB, LRAT Error, Program</td>
</tr>
<tr>
<td>40</td>
<td>ST</td>
<td>Store operation</td>
<td>Alignment, Data Storage, Data TLB, LRAT Error</td>
</tr>
<tr>
<td>41</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>DLK₀</td>
<td>(Implementation-dependent)</td>
<td>(Implementation-dependent)</td>
</tr>
<tr>
<td>43</td>
<td>DLK₁</td>
<td>(Implementation-dependent)</td>
<td>(Implementation-dependent)</td>
</tr>
<tr>
<td>44</td>
<td>AP</td>
<td>Auxiliary Processor operation</td>
<td>Alignment, Data Storage, Data TLB, LRAT Error</td>
</tr>
<tr>
<td>45</td>
<td>PUO</td>
<td>Unimplemented Operation exception</td>
<td>Program</td>
</tr>
<tr>
<td>46</td>
<td>BO</td>
<td>Byte Ordering exception</td>
<td>Data Storage, Instruction Storage</td>
</tr>
<tr>
<td>47</td>
<td>PIE</td>
<td>Imprecise exception</td>
<td>Program</td>
</tr>
<tr>
<td>48:52</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>DATA</td>
<td>Data Access [Category: Embedded.Page Table]</td>
<td>LRAT Error</td>
</tr>
<tr>
<td>54</td>
<td>TLBI</td>
<td>TLB Ineligible [Category: Embedded.Page Table]</td>
<td>Data Storage, Instruction Storage, LRAT Error</td>
</tr>
<tr>
<td>55</td>
<td>PT</td>
<td>Page Table [Category: Embedded.Page Table]</td>
<td>Data Storage, Instruction Storage, LRAT Error</td>
</tr>
<tr>
<td>56</td>
<td>SPV</td>
<td>Signal Processing operation [Category: Signal Processing Engine]</td>
<td>Alignment, Data Storage, Data TLB, LRAT Error, Embedded Floating-point Data, Embedded Floating-point Round, SPE/Embedded Floating-point/Vector Unavailable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vector operation [Category: Vector]</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>EPID</td>
<td>External Process ID operation [Category: Embedded.External Process ID]</td>
<td>Alignment, Data Storage, Data TLB, LRAT Error</td>
</tr>
</tbody>
</table>

The ESR provides a syndrome to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupts, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other ESR bits are cleared. Other interrupt types do not affect the contents of the ESR. The ESR does not need to be cleared by software. Figure 61 shows the bit definitions for the ESR.
Chapter 7. Interrupts and Exceptions

Figure 61. Exception Syndrome Register Definitions

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Name</th>
<th>Meaning</th>
<th>Associated Interrupt Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>VLEMI</td>
<td>VLE operation [Category: VLE]</td>
<td>Alignment, Data Storage, Data TLB, SPE/Embedded Floating-point/Vector Unavailable, Embedded Floating-point Data, Embedded Floating-point Round, Instruction Storage, LRAT Error, Program, System Call</td>
</tr>
<tr>
<td>59:61</td>
<td>Implementation-dependent</td>
<td>(Implementation-dependent)</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>MIF</td>
<td>Misaligned Instruction [Category: VLE]</td>
<td>Instruction TLB, Instruction Storage</td>
</tr>
</tbody>
</table>

Programming Note

The information provided by the ESR is not complete. System software may also need to identify the type of instruction that caused the interrupt, examine the TLB entry accessed by a data or instruction storage access, as well as examine the ESR to fully determine what exception or exceptions caused the interrupt. For example, a Data Storage interrupt may be caused by both a Protection Violation exception as well as a Byte Ordering exception. System software would have to look beyond ESRBO, such as the state of MSRPR in SRR1 and the page protection bits in the TLB entry accessed by the storage access, to determine whether or not a Protection Violation also occurred.

The contents of the ESR can be read into bits 32:63 of register RT using `mfspr RT,ESR`, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the ESR using `mtspr ESR,RS`.

This register is hypervisor privileged.

7.2.14 Guest Exception Syndrome Register [Category: Embedded.Hypervisor]

The Guest Exception Syndrome Register (GESR) is a 32-bit register. GESR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The GESR is identical in form and function to the ESR, but is updated in place of the ESR when an interrupt is directed to the guest. For a description of bit settings and meanings see Section 7.2.13, "Exception Syndrome Register".

The contents of the GESR can be read into bits 32:63 of register RT using `mfspr RT,GESR`, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the GESR using `mtspr GESR,RS`.

This register is privileged.

Programming Note

`mfspr RT,ESR` should be used to read GESR in guest supervisor state. `mtspr ESR,RS` should be used to write GESR in guest supervisor state. See Section 2.2.1, "Register Mapping".

7.2.15 Interrupt Vector Offset Registers [Category: Embedded.Phased-Out]

The Interrupt Vector Prefix Register (IVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

The IVPR is used for Machine Check interrupt if the MCIIVPR is not supported. The IVPR is used for other interrupts if Category E.HV is not supported or if the interrupt is directed to the hypervisor state. For these interrupts, the IVPR is used in one of the following ways.

- If Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported, the following applies. Bits 48:63 are reserved. Bits 0:47 of the Interrupt Vector Prefix Register provide the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets from the appropriate IVOR (provided in Section 7.2.15) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, the following applies. IVPR52:63 are reserved. Bits 0:51 of the Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the exception processing routines. The 12-bit exception vector offsets (provided in Section 7.2.15) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.
Figure 62. Interrupt Vector Offset Register Assignments

Bits 48:59 of the contents of IVORi can be read into bits 48:59 of register RT using \texttt{mfspr RT,IVORi}, setting bits 0:47 and bits 60:63 of GPR(RT) to zero. Bits 48:59 of the contents of register RS can be written into bits 48:59 of IVORi using \texttt{mtspr IVORi,RS}.

These registers are hypervisor privileged.

\subsection*{7.2.16 Guest Interrupt Vector Offset Register} [Category: Embedded.Hypervisor.Phased-Out]

The Guest Interrupt Vector Offset Registers (GIVORs) are 32-bit registers. Guest Interrupt Vector Offset Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). Bits 32:47 and bits 60:63 are reserved. A Guest Interrupt Vector Offset Register provides the quadword index from the base address provided by the GIVPR (see Section 7.2.12) for its respective guest state interrupt. Guest Interrupt Vector Offset Registers are analogous to Interrupt Vector Offset Registers except that they are used when an interrupt is directed to the guest supervisor state. Figure 63 provides the assignments of specific Guest Interrupt Vector Offset Registers to specific interrupts.

Figure 63. Guest Interrupt Vector Offset Register Assignments

Bits 48:59 of the contents of GIVORi can be read into bits 48:59 of register RT using \texttt{mfspr RT,GIVORi}, setting bits 0:47 and bits 60:63 of GPR(RT) to zero. Bits 48:59 of the contents of register RS can be written into bits 48:59 of GIVORi using \texttt{mtspr GIVORi,RS}.

Write access to these registers is hypervisor privileged. Read access to these registers is privileged.
7.2.17 Logical Page Exception Register [Category: Embedded.Hypervisor and Embedded.Page Table]

The Logical Page Exception Register (LPER) is a 64-bit register that is required when both the Embedded.Hypervisor and Embedded.Page Table categories are supported. LPER bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

The LPER fields are described below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:52</td>
<td>Abbreviated Logical Page Number (ALPN)</td>
</tr>
<tr>
<td></td>
<td>This field contains the Abbreviated Real Page Number from the PTE which caused the LRAT Error interrupt. Only bits corresponding to the PTE_ARPN bits supported by the implementation need be implemented.</td>
</tr>
<tr>
<td>60:63</td>
<td>Logical Page Size (LPS)</td>
</tr>
<tr>
<td></td>
<td>This field contains the Page Size from the PTE that caused the LRAT Error interrupt.</td>
</tr>
</tbody>
</table>

All other fields are reserved.

The LPER contains the values of the ARPN and PS fields from the PTE that was used to translate a virtual address for an instruction fetch, Load, Store or Cache Management instruction that caused an LRAT Error interrupt as a result of an LRAT Miss exception. The contents of LPER are unchanged by an interrupt for any other type of exception.

The LPER is a hypervisor resource.

The contents of the Logical Page Exception Register can be read into register RT using mfspr RT,LPER. The contents of register RS can be written into the LPER using mtspr LPER,RS. On both a 32-bit and a 64-bit implementation, the contents of register RS_{32:63} can be written into the LPER_{0:31} using mtspr LPERU,RS.

On a 32-bit implementation that supports fewer than 33 bits of real address, it is implementation-dependent whether the SPR number for LPERU is treated as a reserved value for mfspr and mtspr.

The LPER is a hypervisor resource.

7.2.18 Machine Check Registers

A set of Special Purpose Registers are provided to support Machine Check interrupts.

7.2.18.1 Machine Check Save/Restore Register 0

Machine Check Save/Restore Register 0 (MCSRR0) is a 64-bit register used to save machine state on Machine Check interrupts, and to restore machine state when an rfmc i is executed. When a Machine Check interrupt is taken, the MCSRR0 is set to the current or next instruction address. When rfmc i is executed, instruction execution continues at the address in MCSRR0.

The contents of MCSRR0 when a Machine Check interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_CM) and the computation mode entered for execution of the Machine Check interrupt (specified by EPCR_CM) [Category:Embedded.Hypervisor]. The contents of MCSRR0 upon Machine Check interrupt can be described as follows (assuming Addr is the address to be put into MCSRR0):

if (MSR_CM = 0) & (EPCR_CM = 0) then MCSRR0 ← 32undefined || Addr_{32:63}
if (MSR_CM = 0) & (EPCR_CM = 1) then MCSRR0 ← 320 || Addr_{32:63}
if (MSR_CM = 1) & (EPCR_CM = 1) then MCSRR0 ← Addr_{0:63}
if (MSR_CM = 1) & (EPCR_CM = 0) then MCSRR0 ← undefined

The contents of MCSRR0 can be read into register RT using mfspr RT,MCSRR0. The contents of register RS can be written into MCSRR0 using mtspr MCSRR0,RS.

This register is hypervisor privileged.

---

**Programming Note**

mfspr RT,IVORi should be used to read GIVORi in guest supervisor state. mtspr IVORi,RS should be used to write GIVOR in guest supervisor state. Hypervisor software should emulate the accesses for the guest.

**Programming Note**

The architecture only provides a few GIVORs that are implemented in hardware that are performance critical. Hypervisor software should emulate access to IVORs that do not have corresponding GIVORs.
7.2.18.2 Machine Check Save/Restore Register 1

Machine Check Save/Restore Register 1 (MCSRR1) is a 32-bit register used to save machine state on Machine Check interrupts, and to restore machine state when an rfmc1i is executed. When a Machine Check interrupt is taken, the contents of the MSR are placed into MCSRR1. When rfmc1i is executed, the contents of MCSRR1 are placed into the MSR.

Bits of MCSRR1 that correspond to reserved bits in the MSR are also reserved.

Figure 66. Machine Check Save/Restore Register 1

The contents of MCSRR1 can be read into register RT using mfspr RT,MCSRR1. The contents of register RS can be written into the MCSRR1 using mtspr MCSRR1,RS.

This register is hypervisor privileged.

7.2.18.3 Machine Check Syndrome Register

MCSR (MCSR) is a 64-bit register that is used to record the cause of the Machine Check interrupt. The specific definition of the contents of this register are implementation-dependent (see the User Manual of the implementation).

The contents of MCSR can be read into register RT using mfspr RT,MCSR. The contents of register RS can be written into the MCSR using mtspr MCSR,RS.

This register is hypervisor privileged.

7.2.18.4 Machine Check Interrupt Vector Prefix Register

The Machine Check Interrupt Vector Prefix Register (MCIVPR) is a 64-bit register. MCIVPR is supported only if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported. Whether the MCIVPR is supported is implementation-dependent.

Machine Check Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). MCIVPR[52:63] are reserved. Bits 0:51 of the Machine Check Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the Machine Check exception processing routine. The 12-bit Machine Check exception vector offset (provided in Section 7.2.15) is concatenated to the right of bits 0:47 of the Machine Check Interrupt Vector Prefix Register to form the 64-bit address of the Machine Check exception processing routine.

The contents of Machine Check Interrupt Vector Prefix Register can be read into register RT using mfspr RT,IVPR. The contents of register RS can be written into Machine Check Interrupt Vector Prefix Register using mtspr IVPR,RS.

Program Note
In some implementations that support Interrupt Fixed Offsets, certain instruction cache errors result in a Machine Check exception. The Machine Check interrupt handler needs to be in Caching Inhibited storage in order for the interrupt handler to operate despite an instruction cache error.

7.2.19 External Proxy Register

[Category: External Proxy]

The External Proxy Register (EPR) contains implementation-dependent information related to an External Input interrupt when an External Input interrupt occurs. The EPR is only considered valid from the time that the External Input Interrupt occurs until MSR[EE] is set to 1 as the result of a mtmsr or a return from interrupt instruction.

The format of the EPR is shown below.

Figure 67. External Proxy Register

When the External Input interrupt is taken, the contents of the EPR provide information related to the External Input Interrupt.

This register is hypervisor privileged.

Program Note
The EPR is provided for faster interrupt processing as well as situations where an interrupt must be taken, but software must delay the resultant processing for later.

The EPR contains the vector from the interrupt controller. The process of receiving the interrupt into the EPR acknowledges the interrupt to the interrupt controller. The method for enabling or disabling the acknowledgment of the interrupt by placing the interrupt-related information in the EPR is implementation-dependent. If this acknowledgement is disabled, then the EPR is set to 0 when the External Input interrupt occurs.
7.2.20 Guest External Proxy Register [Category: Embedded Hypervisor, External Proxy]

The Guest External Proxy Register (GEPR) contains implementation-dependent information related to an External Input interrupt when an External Input interrupt directed to the guest occurs. The GEPR is only considered valid from the time that the External Input Interrupt occurs until MSR_EE is set to 1 as the result of a `mtmsr` or a return from interrupt instruction.

The format of the GEPR is shown below.

![GEPR Format](image)

Figure 68. Guest External Proxy Register

When the External Input interrupt is taken in the guest supervisor state, the contents of the GEPR provide information related to the External Input Interrupt.

The contents of the GEPR can be read into bits 32:63 of register RT using `mfspr RT,GEPR`, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the GEPR using `mtspr GEPR,RS`.

The GEPR is identical in form and function to the EPR.

This register is privileged.

---

**Programming Note**

Writing the GEPR register is allowed from both guest supervisor state and hypervisor state. Hypervisor must be able to write GEPR to virtualize External Input interrupt handling for the guest if the guest is using External Proxy. Writing to EPR from the guest is not mapped and results in the same behavior as any undefined supervisor level SPR.

---

**Programming Note**

The GEPR is provided for faster interrupt processing as well as situations where an interrupt must be taken, but software must delay the resultant processing for later.

The GEPR contains the vector from the interrupt controller. The process of receiving the interrupt into the GEPR acknowledges the interrupt to the interrupt controller. The method for enabling or disabling the acknowledgment of the interrupt by placing the interrupt-related information in the GEPR is implementation-dependent. If this acknowledgement is disabled, then the GEPR is set to 0 when the External Input interrupt occurs.

**Programming Note**

`mfspr RT,EPR` should be used to read GEPR in guest supervisor state. Hypervisor software should emulate the accesses for the guest. This keeps the programming model consistent for an operating system running as a guest and running directly in hypervisor state.
7.3 Exceptions

There are two kinds of exceptions, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several types of interrupts to be invoked.

Examples of exceptions that can be caused directly by the execution of an instruction include but are not limited to the following:

- an attempt to execute a reserved-illegal instruction (Illegal Instruction exception type Program interrupt)
- an attempt by an application program to execute a "privileged" instruction (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a "privileged" Special Purpose Register (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a Special Purpose Register that does not exist (Unimplemented Operation Instruction exception type Program interrupt)
- an attempt by a system program to access a Special Purpose Register that does not exist (boundedly undefined results)
- the execution of a defined instruction using an invalid form (Illegal Instruction exception type Program interrupt, Unimplemented Operation exception type Program interrupt, or Privileged Instruction exception type Program interrupt)
- an attempt to access a storage location that is either unavailable (Instruction TLB Error interrupt or Data TLB Error interrupt) or not permitted (Instruction Storage interrupt or Data Storage interrupt)
- an attempt to access storage with an effective address alignment not supported by the implementation (Alignment interrupt)
- the execution of a System Call instruction (System Call interrupt)
- the execution of a Trap instruction whose trap condition is met (Trap type Program interrupt)
- the execution of a floating-point instruction when floating-point instructions are unavailable (Floating-point Unavailable interrupt)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (Enabled exception type Program interrupt)
- the execution of a defined instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an auxiliary processor instruction when the auxiliary processor instruction is unavailable (Auxiliary Processor Unavailable interrupt)
- the execution of an instruction that causes an auxiliary processor enabled exception (Enabled exception type Program interrupt)

The invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the Floating-point Enabled Exception type Program interrupt is in effect then the invocation of the Floating-point Enabled Exception type Program interrupt may be imprecise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the interrupt, has not yet occurred).

7.4 Interrupt Classification

All interrupts, except for Machine Check, can be classified as either Asynchronous or Synchronous. Independent from this classification, all interrupts, including Machine Check, can be classified into one of the following classes:

- Guest [Category:Embedded.Hypervisor]
- Base
- Critical
- Machine Check
- Debug[Category:Embedded.Enhanced Debug].

7.4.1 Asynchronous Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported to the exception handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

7.4.2 Synchronous Interrupts

Synchronous interrupts are those that are caused directly by the execution (or attempted execution) of instructions, and are further divided into two classes, precise and imprecise.

Synchronous, precise interrupts are those that precisely indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction.

Synchronous, imprecise interrupts are those that may indicate the address of the instruction causing the
exception that generated the interrupt, or some instruction after the instruction causing the exception.

### 7.4.2.1 Synchronous, Precise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point.

- **GSRR0** [Category: Embedded.Hypervisor], **SRR0**, **CSR0**, or **DSRR0** [Category: Embedded.Enhanced Debug] addresses either the instruction causing the exception or the instruction immediately following the instruction causing the exception. Which instruction is addressed can be determined from the interrupt type and status bits.

- An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread. However, some storage accesses associated with these preceding instructions may not have been performed with respect to other threads and mechanisms.

- The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type. See Section 7.7 on page 1186.

- Architecturally, no subsequent instruction has executed beyond the instruction causing the exception.

### 7.4.2.2 Synchronous, Imprecise Interrupts

When the execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

When the execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

- **GSRR0** [Category: Embedded.Hypervisor], **SRR0**, or **CSR0** addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.

- An interrupt is generated such that all instructions preceding the instruction addressed by GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSR0 appear to have completed with respect to the executing thread.

- If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception that generates an interrupt (e.g., Alignment, Data Storage), then **GSRR0** [Category: Embedded.Hypervisor] or **SRR0** addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see Section 7.7 on page 1186).

- If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than **sync** or **isync**, then GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by an **sync** or **isync** instruction, then GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSR0 may address either the **sync** or **isync** instruction, or the following instruction.

- If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSR0 may have been partially executed (see Section 7.7 on page 1186).

- No instruction following the instruction addressed by GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSR0 has executed.

### 7.4.3 Interrupt Classes

Interrupts can also be classified as guest [Category: Embedded.Hypervisor], base, critical, Machine Check, and Debug [Category: Embedded.Enhanced Debug].

Interrupt classes other than the guest [Category: Embedded.Hypervisor] or base class may demand immediate attention even if another class of interrupt is currently being processed and software has not yet had the opportunity to save the state of the machine (i.e. return address and captured state of the MSR). For this reason, the interrupts are organized into a hierarchy (see Section 7.8). To enable taking a critical, Machine Check, or Debug [Category: Embedded.Enhanced Debug] interrupt immediately after a guest [Category: Embedded.Hypervisor] or base class interrupt occurs (i.e. before software has saved the state of the machine), these interrupts use the Save/Restore Register pair **CSRR0/CSR1**, **MCSRR0/MCSRR1**, or **DSRR0/DSRR1** [Category: Embedded.Enhanced Debug], and guest [Category: Embedded.Hypervisor] and base class interrupts use Save/Restore Register pairs **GSRR0/GSRR1** and **SRR0/SRR1** respectively.

### 7.4.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until long after the thread has executed.
past the instruction that caused the Machine Check. As such, Machine Check interrupts cannot properly be thought of as synchronous or asynchronous, nor as precise or imprecise. The following general rules apply to Machine Check interrupts:

1. No instruction after the one whose address is reported to the Machine Check interrupt handler in MCSRR0 has begun execution.

2. The instruction whose address is reported to the Machine Check interrupt handler in MCSRR0, and all prior instructions, may or may not have completed successfully. All those instructions that are ever going to complete appear to have done so already, and have done so within the context existing prior to the Machine Check interrupt. No further interrupt (other than possible additional Machine Check interrupts) will occur as a result of those instructions.

### 7.5 Interrupt Processing

Associated with each kind of interrupt is an interrupt vector, that is the address of the initial instruction that is executed when the corresponding interrupt occurs.

When Category: Embedded.Hypervisor is implemented, interrupts are directed (see Section 2.3.1, “Directed Interrupts”) to the guest supervisor state or the hypervisor state, which effects how some MSR bits are set. The conditions under which a given interrupt is directed to the guest supervisor state or hypervisor state is more fully described in the interrupt definitions for each interrupt in Section 7.6, “Interrupt Definitions”.

Interrupt processing consists of saving a small part of the thread's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt can be taken, the following actions are performed, in order:

1. GSRR0 [Category: Embedded.Hypervisor], SRR0, DSRR0 [Category: Embedded.Enhanced Debug], MCSRR0, or CSRR0 is loaded with an instruction address that depends on the interrupt; see the specific interrupt description for details.

2. The GESR [Category: Embedded.Hypervisor] or ESR is loaded with information specific to the exception. Note that many interrupts can only be caused by a single kind of exception event, and thus do not need nor use an ESR setting to indicate to the cause of the interrupt was.

3. GSRR1 [Category: Embedded.Hypervisor], SRR1, DSRR1 [Category: Embedded.Enhanced Debug], or MCSRR1, or CSRR1 is loaded with a copy of the contents of the MSR.

4. The MSR is updated as described below. The new values take effect beginning with the first instruction following the interrupt. MSR bits of particular interest are the following.
   - MSR_PEE,PR,PF,FE,IS,DS,SPV are set to 0 by all interrupts.
   - If Category E.HV is supported, MSR_GS is left unchanged when an interrupt is directed to the guest supervisor state, otherwise they are set to 0 by all interrupts.
   - If Category E.HV is supported, MSR_PPM is left unchanged when an interrupt is directed to the guest supervisor state and MSR_PPMP = 1, otherwise MSR_PPM is set to 0 by all interrupts.
   - If Category E.HV is supported, MSR_UCLE is left unchanged when an interrupt is directed to the guest supervisor state and MSR_UCLEP = 1, otherwise MSR_UCLE is set to 0 by all interrupts.
   - MSR_ME is set to 0 by Machine Check interrupts and left unchanged by all other interrupts.
   - MSR_CE is set to 0 by critical class interrupts, Debug interrupts, and Machine Check interrupts, and is left unchanged by all other interrupts.
   - MSR_DIE is set to 0 by critical class interrupts unless Category E.ED is supported, by Debug interrupts, and by Machine Check interrupts, and is left unchanged by all other interrupts.
   - If Category E.HV is supported and the interrupt is directed to the guest supervisor state, MSR_CM is set to EPCR_GICM, otherwise MSR_CM is set to EPCR_ICM.
   - Other supported MSR bits are left unchanged by all interrupts.

See Section 4.2.1 for more detail on the definition of the MSR.

5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt. If Category E.HV is supported, and the interrupt is directed to the guest state, the location is one of the following, where IVORi (GIVORi) is the (Guest) Interrupt Vector Offset Register for that interrupt (see Figure 63 on page 1152):
   - GIVPR[0:47] || GIVOR[48:59] || 0b0000 if Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported
   - GIVPR[0:51] || fixed offset shown in Figure Figure 62 on page 1152 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported

Otherwise, the location is one of the following:
   - IVPR[0:47] || IVOR[48:59] || 0b0000 if Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported
IVPR0..51 || fixed offset shown in Figure Figure 62 on page 1152 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported and either MCIVPR is not supported or the interrupt is not a Machine Check

MCIVPR0..51 || fixed offset shown in Figure Figure 70 on page 1165 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, MCIVPR is supported, and the interrupt is a Machine Check

The contents of the (Guest) Interrupt Vector Prefix Register, Machine Check Interrupt Vector Prefix Register, and (Guest) Interrupt Vector Offset Registers are indeterminate upon power-on reset, and must be initialized by system software using the \texttt{mtspr} instruction.

It is implementation-dependent whether interrupts clear reservations obtained with \texttt{lbarx}, \texttt{lharx}, \texttt{lwarx}, \texttt{ldarx}, or \texttt{lqarx}.

Interrupts might not clear reservations obtained with \texttt{Load and Reserve} instructions. The operating system or hypervisor should do so at appropriate points, such as at process switch or a partition switch.

At the end of an interrupt handling routine, execution of an \texttt{rfgi} [Category: Embedded.Hypervisor], \texttt{rfi}, \texttt{rfdi} [Category: Embedded.Enhanced Debug], \texttt{rfmci}, or \texttt{rfci} causes the MSR to be restored from the contents of GSRR1 [Category: Embedded.Hypervisor], SRR1, DSRR1 [Category: Embedded.Enhanced Debug], MCSRR1, or CSRR1, and instruction execution to resume at the address contained in GSRR0 [Category: Embedded.Hypervisor], SRR0, DSRR0 [Category: Embedded.Enhanced Debug], MCSRR0, or CSRR0, respectively.

**Programming Note**

In general, at process switch (partition switch), due to possible process interlocks and possible data availability requirements, the operating system (hypervisor) needs to consider executing the following.

- \texttt{stbcx, sthcx, stwcx, stdcx,} or \texttt{stqcx}, to clear the reservation if one is outstanding, to ensure that a \texttt{lbarx}, \texttt{lharx}, \texttt{lwarx}, \texttt{ldarx}, or \texttt{lqarx} in the “old” process (partition) is not paired with a \texttt{stbcx}, \texttt{sthcx}, \texttt{stwcx}, \texttt{stdcx}, or \texttt{stqcx} in the “new” process (partition).

- \texttt{sync}, to ensure that all storage operations of an interrupted process are complete with respect to other threads before that process begins executing on another thread.

- \texttt{isync}, \texttt{rfgi <E.HV>, rfi, rfdi} [Category: Embedded.Enhanced Debug], \texttt{rfmci}, or \texttt{rfci} to ensure that the instructions in the “new” process execute in the “new” context.
Programming Note

For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:
- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by `lmw` for which the storage operand is not aligned, or by `dcbz` or `dcbzep` for which the storage operand is in storage that is Write Through Required or Caching Inhibited.
- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: Illegal Instruction type Program interrupt caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports, or by an instruction that is in a category that the implementation does not support but is used by some programs that the operating system supports.

In general, the instruction should not be emulated if:
- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by `sc`.
- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by `lwarx` for which the storage operand is not aligned.
- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by `lwz` for which the storage operand is in storage that the program should be permitted to access but for which there currently is no TLB entry.)
## 7.6 Interrupt Definitions

Table 69 provides a summary of each interrupt type, the various exception types that may cause that interrupt type, the classification of the interrupt, which ESR (GESR) bits can be set, if any, which MSR bits can mask the interrupt type and which Interrupt Vector Offset Register is used to specify that interrupt type’s vector address.

<table>
<thead>
<tr>
<th>IVOR</th>
<th>Interrupt</th>
<th>Exception</th>
<th>Asynchronous</th>
<th>Synchronous, Precise, Imprecise, Critical</th>
<th>ESR (GESR) (See Note 5)</th>
<th>MSR Mask Bit(s)¹</th>
<th>Category (Section 1.3.5 of Book I)</th>
<th>Notes (see page 1164)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVOR0</td>
<td>Critical Input</td>
<td>Critical Input</td>
<td>x</td>
<td>x</td>
<td></td>
<td>CE or GS ME</td>
<td>E 1 1165</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR1</td>
<td>Machine Check</td>
<td>Machine Check</td>
<td></td>
<td></td>
<td>[ST],[FP,AP,SPV],[PT], [VLEMI],[EPI</td>
<td>[ST],[VLEMI]</td>
<td>E 2,4 1165</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR2</td>
<td>Data Storage</td>
<td>Access</td>
<td>x</td>
<td>[ST],[FP,AP,SPV],[PT], [VLEMI],[EPI</td>
<td>[ST],[VLEMI]</td>
<td></td>
<td>E 9 1166</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load and Reserve or</td>
<td>x</td>
<td>[DLK0,DLK1],[ST]</td>
<td></td>
<td></td>
<td>E 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store Conditional to</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘write-thru required’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>storage (W=1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cache Locking</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte Ordering</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtualization Fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.PT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Page Table Fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.PT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB Ineligible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.PT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte Ordering</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mismatched Instruction</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Storage (See Book VLE.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mismatched Instruction</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>MIF</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Storage (See Book VLE.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Page Table Fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.PT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB Ineligible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.PT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ MSR Mask Bit(s) can be set if the corresponding ESR (GESR) bit(s) are set.

[Page 1165]
<table>
<thead>
<tr>
<th>IVOR</th>
<th>Interrupt</th>
<th>Exception</th>
<th>Asynchronous, Precise, Synchronous, Imprecise, Critical</th>
<th>ESR (GESR) (See Note 5)</th>
<th>MSR Mask Bit(s)</th>
<th>DBCR0/TCR Mask Bit</th>
<th>Category (Section 1.3.5 of Book I)</th>
<th>Notes (see page 1164)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVOR4</td>
<td>External Input</td>
<td>External Input</td>
<td>x</td>
<td>E</td>
<td>EE or GS</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1170</td>
</tr>
<tr>
<td>GIVOR4</td>
<td>External Input</td>
<td>External Input</td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1170</td>
</tr>
<tr>
<td>IVOR5</td>
<td>Alignment</td>
<td>Alignment</td>
<td>x</td>
<td>[ST],[FP,AP,SPV]</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1171</td>
</tr>
<tr>
<td>IVOR6</td>
<td>Program</td>
<td>Illegal</td>
<td>x</td>
<td>PIL, [VLEMI]</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1172</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Privileged</td>
<td>x</td>
<td>PPR,[AP], [VLEMI]</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1172</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trap</td>
<td>x</td>
<td>PTR,[VLEMI]</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1172</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP Enabled</td>
<td>x</td>
<td>FP, [PIE]</td>
<td>FE0, FE1</td>
<td>E</td>
<td></td>
<td></td>
<td>1172</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AP Enabled</td>
<td>x</td>
<td>AP</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1172</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unimplemented Op</td>
<td>x</td>
<td>PUO, [VLEMI]</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1172</td>
</tr>
<tr>
<td>IVOR7</td>
<td>FP Unavailable</td>
<td>FP Unavailable</td>
<td>x</td>
<td>[VLEMI]</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td>1173</td>
</tr>
<tr>
<td>IVOR8</td>
<td>System Call</td>
<td>System Call</td>
<td>x</td>
<td>[VLEMI]</td>
<td>E</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1173</td>
</tr>
<tr>
<td>GIVOR8</td>
<td>(E.HV)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR9</td>
<td>AP Unavailable</td>
<td>AP Unavailable</td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1173</td>
</tr>
<tr>
<td>IVOR10</td>
<td>Decrementer</td>
<td>AP Unavailable</td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>E</td>
<td></td>
<td></td>
<td>1174</td>
</tr>
<tr>
<td>GIVOR10</td>
<td>(E.HV)</td>
<td>Guest Decrementer</td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1174</td>
</tr>
<tr>
<td>IVOR11</td>
<td>FIT</td>
<td></td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>FIE</td>
<td></td>
<td></td>
<td>1175</td>
</tr>
<tr>
<td>GIVOR11</td>
<td>(E.HV)</td>
<td>Guest FIT</td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>FIE</td>
<td></td>
<td></td>
<td>1175</td>
</tr>
<tr>
<td>IVOR12</td>
<td>Watchdog</td>
<td></td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>WIE</td>
<td></td>
<td></td>
<td>1176</td>
</tr>
<tr>
<td>GIVOR12</td>
<td>(E.HV)</td>
<td>Guest Watchdog</td>
<td>x</td>
<td>E</td>
<td>EE and GS</td>
<td>WIE</td>
<td></td>
<td></td>
<td>1176</td>
</tr>
<tr>
<td>IVOR13</td>
<td>Data TLB Error</td>
<td>Data TLB Miss</td>
<td>x</td>
<td>[ST],[FP,AP,SPV]</td>
<td>E</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1177</td>
</tr>
<tr>
<td>GIVOR13</td>
<td>(E.HV)</td>
<td></td>
<td></td>
<td>[VLEMI],[EPIID]</td>
<td>E</td>
<td>E.HV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR14</td>
<td>Inst TLB Error</td>
<td>Inst TLB Miss</td>
<td>x</td>
<td>[MIF]</td>
<td>E</td>
<td>E.HV</td>
<td></td>
<td></td>
<td>1178</td>
</tr>
<tr>
<td>GIVOR14</td>
<td>(E.HV)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR</td>
<td>Interrupt</td>
<td>Exception</td>
<td>Asynchronous</td>
<td>Synchronous, Precise</td>
<td>ESR (GESR) (See Note 5)</td>
<td>MSR Mask Bit(s)</td>
<td>DBC/TrCR Mask Bit</td>
<td>Category (Section 1.3.5 of Book I)</td>
<td>Notes (see page 1164)</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------</td>
<td>----------------------------</td>
<td>--------------</td>
<td>----------------------</td>
<td>------------------------</td>
<td>----------------</td>
<td>-------------------</td>
<td>------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>IVOR15</td>
<td>Debug</td>
<td>Trap</td>
<td>x</td>
<td>x</td>
<td>DE, IDM</td>
<td>E</td>
<td>10</td>
<td>1178</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inst Addr Compare</td>
<td></td>
<td>x</td>
<td>x</td>
<td>DE, IDM</td>
<td>E</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data Addr Compare</td>
<td></td>
<td>x</td>
<td>x</td>
<td>DE, IDM</td>
<td>E</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction Complete</td>
<td></td>
<td>x</td>
<td>x</td>
<td>DE, IDM</td>
<td>E</td>
<td>3,10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch Taken</td>
<td></td>
<td>x</td>
<td>x</td>
<td>DE, IDM</td>
<td>E</td>
<td>3,10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return From Interrupt</td>
<td></td>
<td>x</td>
<td>x</td>
<td>DE, IDM</td>
<td>E</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interrupt Taken</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncond Debug Event</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Critical Interrupt Taken</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Critical Interrupt Return</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR32</td>
<td>SPE/Embedded Floating-Point/Vector Unavailable</td>
<td>SPE Unavailable</td>
<td>x</td>
<td>SPV, [VLEMI]</td>
<td>SPE</td>
<td>1179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR33</td>
<td>Embedded Floating-Point Data</td>
<td>Embedded Floating-Point Data</td>
<td>x</td>
<td>SPV, [VLEMI]</td>
<td>SP.P*</td>
<td>1180</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR34</td>
<td>Embedded Floating-Point Round</td>
<td>Embedded Floating-Point Round</td>
<td>x</td>
<td>SPV, [VLEMI]</td>
<td>SP.P*</td>
<td>1180</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR35</td>
<td>Embedded Performance Monitor</td>
<td>Embedded Performance Monitor</td>
<td>x</td>
<td>EE or GS, E.PM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GIVOR35</td>
<td>Embedded Performance Monitor</td>
<td>Embedded Performance Monitor</td>
<td>x</td>
<td>EE or GS, E.PM, E.HV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR36</td>
<td>Processor Doorbell</td>
<td>Processor Doorbell</td>
<td>x</td>
<td>EE or GS</td>
<td>E.PC</td>
<td></td>
<td>1181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR37</td>
<td>Processor Doorbell Critical</td>
<td>Processor Doorbell Critical</td>
<td>x</td>
<td>CE or GS</td>
<td>E.PC</td>
<td></td>
<td>1183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR38</td>
<td>Guest Processor Doorbell</td>
<td>Guest Processor Doorbell</td>
<td>x</td>
<td>EE or GS</td>
<td>E.PC, E.HV</td>
<td></td>
<td>1181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR39</td>
<td>Guest Processor Doorbell Critical</td>
<td>Guest Processor Doorbell Critical</td>
<td>x</td>
<td>CE or GS</td>
<td>E.PC, E.HV</td>
<td></td>
<td>1181</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Guest Processor Doorbell Machine Check</td>
<td>Guest Processor Doorbell Machine Check</td>
<td>x</td>
<td>ME or GS</td>
<td>E.PC, E.HV</td>
<td></td>
<td>1183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR40</td>
<td>Embedded Hypervisor System Call</td>
<td>Embedded Hypervisor System Call</td>
<td>x</td>
<td>[VLEMI]</td>
<td>E.HV</td>
<td></td>
<td>1183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR41</td>
<td>Embedded Hypervisor Privilege</td>
<td>Embedded Hypervisor Privilege</td>
<td>x</td>
<td>[VLEMI]</td>
<td>E.HV</td>
<td></td>
<td>1184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVOR42</td>
<td>LRAT Error</td>
<td>LRAT Miss</td>
<td>x</td>
<td>[ST], [FPAP, SPV], [DATA], [PT], [VLEMI], [E.PID]</td>
<td>E.HV, LRAT</td>
<td></td>
<td>1184</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. If an expression of MSR bits is provided, the interrupt is masked if the expression evaluates to 0 and is enabled if the expression evaluates to 1.

Figure 69. Interrupt and Exception Types

Figure 69 Notes

1. Although it is not specified, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of Critical Input and External Input interrupts.

2. Machine Check interrupts are a special case and are not classified as asynchronous nor synchronous. See Section 7.4.4 on page 1157.

3. The Instruction Complete and Branch Taken debug events are only defined for MSRDE=1 when in Internal Debug Mode (DBCR0DM=1). In other words, when in Internal Debug Mode with MSRDE=0, then Instruction Complete and Branch Taken debug events cannot occur, and no DBSR status bits are set and no subsequent imprecise Debug interrupt will occur (see Section 10.4 on page 1213).

4. Machine Check status information is commonly provided as part of the system implementation, but is implementation-dependent.

5. In general, when an interrupt causes a particular ESR (GESR) bit or bits to be set (or cleared) as indicated in the table, it also causes all other ESR (GESR) bits to be cleared. There may be special rules regarding the handling of implementation-specific ESR (GESR) bits.

   Legend:
   
   \[ \text{[xxx]} \] means ESR(GESR)\_xxx could be set
   \[ \text{[xxx,yyy]} \] means either ESR(GESR)\_xxx or ESR(GESR)\_yyy may be set, but never both
   \[ \text{(xxx,yyy)} \] means either ESR(GESR)\_xxx or ESR(GESR)\_yyy will be set, but never both
   \[ \text{(xxx,yyy)} \] means either ESR(GESR)\_xxx or ESR(GESR)\_yyy will be set, or possibly both
   xxx means ESR(GESR)\_xxx is set

6. The precision of the Floating-point Enabled Exception type Program interrupt is controlled by the MSR\_\text{FE}_0,\text{FE}_1 bits. When MSR\_\text{FE}_0,\text{FE}_1=0b01 or 0b10, the interrupt may be imprecise. When such a Program interrupt is taken, if the address saved in SRR0 is not the address of the instruction that caused the exception (i.e. the instruction that caused FPSCR\_\text{FE}X to be set to 1), ESR\_\text{PIE} is set to 1. When MSR\_\text{FE}_0,\text{FE}_1=0b11, the interrupt is precise. When MSR\_\text{FE}_0,\text{FE}_1=0b00, the interrupt is masked, and the interrupt will subsequently occur imprecisely if and when Floating-point Enabled Exception type Program interrupts are enabled by setting either or both of MSR\_\text{FE}_0,\text{FE}_1, and will also cause ESR\_\text{PIE} to be set to 1. See Section 7.6.8. Also, exception status on the exact cause is available in the Floating-Point Status and Control Register (see Section 4.2.2 and Section 4.4 of Book I).

   The precision of the Auxiliary Processor Enabled Exception type Program interrupt is implementation-dependent.

7. Auxiliary Processor exception status is commonly provided as part of the implementation.

8. Cache locking and cache locking exceptions are implementation-dependent.

9. Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.

10. If the Embedded.Enhanced Debug category is enabled, this interrupt is not a critical interrupt. DSRR0 and DSRR1 are used instead of CSRR0 and CSRR1.

7.6.1 Interrupt Fixed Offsets [Category: Embedded.Phased-In]

Figure 62 on page 1152 shows the 12-bit low-order effective address offset for each interrupt type. This value is the offset from the base address provided by either the IVPR (see Section 7.2.11) or the GIVPR (see Section 7.2.12).
7.6.2 Critical Input Interrupt

A Critical Input interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Critical Input exception is presented to the interrupt mechanism, and MSR_CE=1. If Category: Embedded.Hypervisor is supported, Critical Input interrupts with the exception of Guest Processor Doorbell Critical are enabled regardless of the state of MSR_CE when MSR_QS=1. While the specific definition of a Critical Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the system. Also, implementations may provide an alternative means (in addition to MSR_CE) for masking the Critical Input interrupt.

CSRR0, CSRR1, and MSR are updated as follows:

CSRR0, CSRR1, and MSR are updated as follows:

**CSRR0**
- Set to the effective address of the next instruction to be executed.

**CSRR1**
- Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM** set to EPCR_ICM.
- **ME** Unchanged.
- **DE** Unchanged if category E.ED is supported; otherwise set to 0

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Hypervisor] are supported, instruction execution resumes at address IVPR_0:51 || 0x020. Otherwise, instruction execution resumes at address IVPR_0:47 || IVOR_48:59 || 0b0000.

**Programming Note**
Software is responsible for taking any action(s) that are required by the implementation in order to clear any Critical Input exception status prior to re-enabling MSR_CE in order to avoid another, redundant Critical Input interrupt.

7.6.3 Machine Check Interrupt

A Machine Check interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Machine Check exception is presented to the interrupt mechanism, and MSR_ME=1. If Category: Embedded.Hypervisor is supported, Machine Check interrupts with the exception of Guest Processor Doorbell Machine Check are enabled regardless of the state of MSR_ME when MSR_QS=1. The specific cause or causes of Machine Check exceptions are implementation-dependent, as are the details of the actions taken on a Machine Check interrupt.
If the Machine Check Extension is implemented, MCSRR0, MCSRR1, and MCSR are set, otherwise CSRR0, CSRR1, and ESR are set. The registers are updated as follows:

**CSRR0/MCSRR0**
Set to an instruction address. As closely as possible, set to the effective address of an instruction that was executing or about to be executed when the Machine Check exception occurred.

**CSRR1/MCSRR1**
Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM** \(\text{MSR}_{\text{CM}}\) is set to \(\text{EPCR}_{\text{CM}}\).
- All other defined MSR bits set to 0.

**ESR/MCSR**
Implementation-dependent.

Instruction execution resumes at address \(\text{IVPR}_{0:47} \| \text{IVOR}_{148:59} \| 0b0000\).

If the Embedded.Hypervisor category is supported, a Machine Check interrupt caused by an error in the storage subsystem, the storage subsystem may return incorrect data, that may be placed into registers and/or on-chip caches.

### 7.6.4 Data Storage Interrupt

A Data Storage interrupt may occur when no higher priority exception exists (see Section 7.9 on page 1190) and a Data Storage exception is presented to the interrupt mechanism. A Data Storage exception is caused when any of the following exceptions arises during execution of an instruction:

#### Read Access Control exception

A Read Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR\(\text{PR}=1\)), a Load or ‘load-class’ Cache Management instruction attempts to access a location in storage that is not user mode read enabled (i.e. page access control bit \(\text{UR}=0\)).
- While in supervisor mode (MSR\(\text{PR}=0\)), a Load or ‘load-class’ Cache Management instruction attempts to access a location in storage that is not supervisor mode read enabled (i.e. page access control bit \(\text{SR}=0\)).

#### Write Access Control exception

A Write Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR\(\text{PR}=1\)), a Store or ‘store-class’ Cache Management instruction attempts to access a location in storage that is not user mode write enabled (i.e. page access control bit \(\text{UW}=0\)).
- While in supervisor mode (MSR\(\text{PR}=0\)), a Store or ‘store-class’ Cache Management instruction attempts to access a location in storage that is not
supervisor mode write enabled (i.e. page access control bit SW=0).

**Byte Ordering exception**

A Byte Ordering exception may occur when the implementation cannot perform the data storage access in the byte order specified by the Endian storage attribute of the page being accessed.

**Cache Locking exception**

A Cache Locking exception may occur when the locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent.

**Storage Synchronization exception**

A Storage Synchronization exception will occur when an attempt is made to execute a Load and Reserve or Store Conditional instruction from or to a location that is Write Through Required or Caching Inhibited (if the interrupt does not occur then the instruction executes correctly: see Section 4.4.2 of Book II).

If a stbcx, sthcx, stwcx, stdcx, or stqcx, would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional Store to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

**Page Table Fault exception**

A Page Table Fault exception is caused when a Page Table translation occurs for a data access due to a Load, Store or Cache Management instruction and the Page Table Entry that is accessed is invalid (PTE Valid bit = 0).

**TLB Ineligible exception**

A TLB Ineligible exception is caused when a Page Table translation occurs for a data access due to a Load, Store or Cache Management instruction and any of the following conditions are true:

- The only TLB entries that can be used to hold the translation for the virtual address have IPROT=1
- No TLB array can be loaded from the Page Table for the page size specified by the PTE.
- The PTE_ARPN is treated as an LPN (The Embedded.Hypervisor category is supported) and there is no TLB array that meets all the following conditions.
- The TLB array supports the page size specified by the PTE.
- The TLB array can be loaded from the Page Table (TLBnCFGPT = 1).

If the Embedded.Hypervisor category is supported, an Data Storage interrupt resulting from a TLB Ineligible exception is always directed to hypervisor state regardless of the setting of EPCR_DSIGS.

**Virtualization Fault exception [Category: Embedded.Hypervisor]**

A Virtualization Fault exception will occur when a Load, Store, or Cache Management Instruction attempts to access a location in storage that has the Virtualization Fault (VF) bit set. A Data Storage interrupt resulting from a Virtualization Fault exception is always directed to hypervisor state regardless of the setting of EPCR_DSIGS.

Instructions lswx or stswx with a length of zero, icbt, dcbt, dcbtep, dcbtst, dcbtstep, or dcba cannot cause a Data Storage interrupt, regardless of the effective address.

**Programming Note**

The icbi, icbiep, icbt, icbtls and icbwc instructions are treated as Loads from the addressed byte with respect to address translation and protection. These Instruction Cache Management instructions use MSR_DS, not MSR_IS, to determine translation for their operands. Instruction Storage exceptions and Instruction TLB Miss exceptions are associated with the ‘fetching’ of instructions not with the ‘execution’ of instructions. Data Storage exceptions and Data TLB Miss exceptions are associated with the ‘execution’ of Instruction Cache Management instructions. One exception to the above is that icbtls and icbwc only cause a Data Storage exception if they have neither execute access nor read access.

When a Data Storage interrupt occurs, the thread suppresses the execution of the instruction causing the Data Storage exception.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

<table>
<thead>
<tr>
<th>MSR</th>
<th>Set to the contents of the MSR at the time of the interrupt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>MSR_CM is set to EPCR_ICM.</td>
</tr>
<tr>
<td>DE</td>
<td>Unchanged.</td>
</tr>
</tbody>
</table>

All other defined MSR bits set to 0.

| DEAR | Set to the effective address of a byte that is both within the range of the bytes being accessed by the Storage Access or Cache Management instruction, and within the |

---

Chapter 7. Interrupts and Exceptions 1167
page whose access caused the Data Storage exception.

ESR

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

ST Set to 1 if the instruction causing the interrupt is a Store or ‘store-class’ Cache Management instruction; otherwise set to 0.

DLK0:1 Set to an implementation-dependent value due to a Cache Locking exception causing the interrupt.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

BO Set to 1 if the instruction caused a Byte Ordering exception; otherwise set to 0.

TLBI Set to 1 if a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt; otherwise set to 0.

PT If a Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, then PT is set to 1 if no TLB entry was created from the Page Table and is set to an implementation-dependent value if a TLB entry was created. See Section 6.7.4 for rules about TLB updates. If no Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, set to 0.

SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state GSRR0, GSRR1, GDEAR, and GESR are set in place of SRR0, SRR1, DEAR, and ESR, respectively. The MSR is set as follows:

MSR

CM MSR_CM is set to EPCR_GICM.

CE, ME, GS, DE Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

The following is a prioritized listing of the various exceptions which cause a Data Storage interrupt and the corresponding ESR bit, if applicable. Even though multiple of these exceptions may occur, at most one of the following exceptions is reported in the ESR.

1. Cache Locking <E.CL>: DLK0:1
2. Page Table Fault <E.PT>: PT
3. Virtualization Fault <E.HV>
4. TLB Ineligible <E.PT>: TLBI
5. Byte Ordering: BO
6. Read Access or Write Access: If the exception occurred during a Page Table translation, PT <E.PT>

Programming Note

Since some Data Storage exceptions are not mutually exclusive, system software may need to examine the TLB entry or the Page Table entry accessed by the data storage access in order to determine whether additional exceptions may have also occurred.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following:

- GIVPR0:47 || GIVOR248:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- GIVPR0:51||0x060 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following:

- IVPR0:47 || IVOR248:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- IVPR0:51||0x060 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an Instruction Storage exception is presented to the interrupt mechanism. An Instruction Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Execute Access Control exception

An Execute Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_PR=1), an instruction fetch attempts to access a location in storage that is not user mode execute enabled (i.e. page access control bit UX=0).
- While in supervisor mode (MSR_PR=0), an instruction fetch attempts to access a location in storage
that is not supervisor mode execute enabled (i.e. page access control bit SX=0).

**Byte Ordering exception**

A Byte Ordering exception may occur when the implementation cannot perform the instruction fetch in the byte order specified by the Endian storage attribute of the page being accessed.

**Page Table Fault exception**

A Page Table Fault exception is caused when a Page Table translation occurs for an instruction fetch and the Page Table Entry that is accessed is invalid (Valid bit = 0).

**TLB Ineligible exception**

A TLB Ineligible exception is caused when a Page Table translation occurs for an instruction fetch and any of the following conditions are true.

- The only TLB entries that can be used to hold the translation for the virtual address have IPROT=1
- No TLB array can be loaded from the Page Table for the page size specified by the PTE.
- The PTEARPN is treated as an LPN (The Embedded.Hypervisor category is supported) and there is no TLB array that meets all the following conditions.
  - The TLB array supports the page size specified by the PTE.
  - The TLB array can be loaded from the Page Table (TLBnCFGPT = 1).

If the Embedded.Hypervisor category is supported, an Instruction Storage interrupt resulting from a TLB Ineligible exception is always directed to hypervisor state regardless of the setting of EPCRISIGS.

When an Instruction Storage interrupt occurs, the thread suppresses the execution of the instruction causing the Instruction Storage exception.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, MSR, and ESR are updated as follows:

**SRR0**

Set to the effective address of the instruction causing the Instruction Storage interrupt.

**SRR1**

Set to the contents of the MSR at the time of the interrupt.

**MSR**

- **CM**
  - MSR<sub>CM</sub> is set to EPCR<sub>ICM</sub>
  - CE, ME, DE
  - Unchanged.

- All other defined MSR bits set to 0.

**ESR**

- **BO**
  - Set to 1 if the instruction fetch caused a Byte Ordering exception; otherwise set to 0.

- **TLBI**
  - Set to 1 if a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt; otherwise set to 0.

- **PT**
  - If a Page Table Fault or an Execute Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, then PT is set to 1 if no TLB entry was created from the Page Table and is set to an implementation-dependent value if a TLB entry was created. See Section 6.7.4 for rules about TLB updates. If no Page Table Fault or Execute Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, set to 0.

- **VLEMI**
  - Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state, GSRR0, GSRR1, and GESR are set in place of SRR0, SRR1, and ESR, respectively. The MSR is set as follows:

**MSR**

- **CM**
  - MSR<sub>CM</sub> is set to EPCR<sub>GICM</sub>
- CE, ME, GS, DE
  - Unchanged.

- Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

The following is a prioritized listing of the various exceptions which cause a Instruction Storage interrupt and the corresponding ESR bit, if applicable. Even though multiple of these exceptions may occur, at most one of the following exceptions is reported in the ESR.

1. Page Table Fault <E.PT>: PT
2. TLB Ineligible <E.PT>: TLBI
3. Byte Ordering exception: BO
4. Execute Access: If the exception occurred during a Page Table translation, PT <E.PT>

---

**Programming Note**

Since some Instruction Storage exceptions are not mutually exclusive, system software may need to examine the TLB entry or the Page Table entry accessed by the data storage access in order to determine whether additional exceptions may have also occurred.
If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.
- GIVPR0:47 || GIVOR348:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- GIVPR0:51||0x080 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.
- IVPR0:47 || IVOR348:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- IVPR0:51||0x080 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

### 7.6.6 External Input Interrupt

An External Input interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), an External Input exception is presented to the interrupt mechanism, and the interrupt is enabled. While the specific definition of an External Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the processing system. Also, implementations may provide an alternative means (in addition to the enabled criteria) for masking the External Input interrupt.

If Category: Embedded.Hypervisor is supported, External Input interrupts are enabled if:

(EPCR<sub>EXTGS</sub> = 0) & ((MSR<sub>GS</sub> =1) | (MSR<sub>EE</sub>=1))

or

(EPCR<sub>EXTGS</sub> = 1) & (MSR<sub>GS</sub> =1) & (MSR<sub>EE</sub>=1)

Otherwise, External Input interrupts are enabled if MSR<sub>EE</sub>=1.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, and MSR are updated as follows:

**SRR0** Set to the effective address of the next instruction to be executed.

**SRR1** Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM** MSR<sub>CM</sub> is set to EHSR<sub>ICM</sub>.
- **CE** Unchanged.
- **ME** Unchanged.
- **GS** Unchanged.
- **DE** Unchanged.

All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to the guest supervisor state, GSRR0 and GSRR1 are set in place of SRR0 and SRR1, respectively. The MSR is set as follows:

**MSR**
- **CM** MSR<sub>CM</sub> is set to EPCR<sub>GICM</sub>.
- **CE** ME,GS,DE Unchanged.

Bits in the MSR corresponding to set bits in the MSR register are left unchanged.

All other defined MSR bits set to 0.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.
- GIVPR0:47 || GIVOR448:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- GIVPR0:51||0x0A0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.
- IVPR0:47 || IVOR448:59||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- IVPR0:51||0x0A0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

---

**Programming Note**

Software is responsible for taking whatever action(s) are required by the implementation in order to clear any External Input exception status prior to re-enabling MSR<sub>EE</sub> in order to avoid another, redundant External Input interrupt.

### 7.6.7 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an Alignment exception is presented to the interrupt mechanism. An Alignment exception may be caused when the implementation cannot perform a data storage access for one of the following reasons:

- The operand of a single-register Load or Store is not aligned.
- The instruction is a Load Multiple or Store Multiple, or a Move Assist for which the length of the storage operand is not zero.
- The operand of dcbz or dcbzep is in storage that is Write Through Required or Caching Inhibited, or one of these instructions is executed in an implementation that has either no data cache or a Write Through data cache or the line addressed by the instruction cannot be established in the cache because the cache is disabled or locked.
- The operand of a Store, except Store Conditional, or Store String for which the length of the storage operand is zero, is in storage that is Write-Through Required.

For **lmw** and **stmw** with an operand that is not word-aligned, and for **Load and Reserve** and **Store**
Conditional instructions with an operand that is not aligned, an implementation may yield boundedly undefined results instead of causing an Alignment interrupt. A Store Conditional to Write Through Required storage may either cause a Data Storage interrupt, cause an Alignment interrupt, or correctly execute the instruction. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an Alignment interrupt. (For dcbz or dcbzep, 'correct' execution means setting each byte of the block in main storage to 0x00.)

When an Alignment interrupt occurs, the thread suppresses the execution of the instruction causing the Alignment exception.

**Programming Note**

The architecture does not support the use of an unaligned effective address by Load and Reserve and Store Conditional instructions. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

When an Alignment interrupt occurs, the thread suppresses the execution of the instruction causing the Alignment exception.

**SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:**

**SRR0**
Set to the effective address of the instruction causing the Alignment interrupt.

**SRR1**
Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM**
  MSR_{CM} is set to EPCR_{ICM}.
- **CE, ME, DE**
  Unchanged
- **CE, ME, DE, ICM**
  Unchanged.

All other defined MSR bits set to 0.

**DEAR**
Set to the effective address of a byte that is both within the range of the bytes being accessed by the Storage Access or Cache Management instruction, and within the page whose access caused the Alignment exception.

**ESR**
- **FP**
  Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
- **ST**
  Set to 1 if the instruction causing the interrupt is a Store; otherwise set to 0.
- **AP**
  Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

**SPV**
Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.

**VLEMI**
Set to 1 if the instruction causing the interrupt resides in VLE storage.

**EPID**
Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded, Phased-In] are supported, instruction execution resumes at address IVPR_{0:51} || 0x0C0. Otherwise, instruction execution resumes at address IVPR_{0:47} || IVORS_{54:59} || 0b0000.

### 7.6.8 Program Interrupt

A Program interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Program exception is presented to the interrupt mechanism, and, for Floating-point Enabled exception, MSR_{FE0, FE1} are non-zero. A Program exception is caused when any of the following exceptions arises during execution of an instruction:

#### Floating-point Enabled exception

A Floating-point Enabled exception is caused when FPSCR_{FEX} is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a Move To FPSCR instruction that causes an exception bit and the corresponding enable bit both to be 1. Note that in this context, the term ‘enabled exception’ refers to the enabling provided by control bits in the Floating-Point Status and Control Register. See Section 4.2.2 of Book I.

#### Auxiliary Processor Enabled exception

The cause of an Auxiliary Processor Enabled exception is implementation-dependent.

#### Illegal Instruction exception

An Illegal Instruction exception does occur when execution is attempted of any of the following kinds of instructions.

- a reserved-illegal instruction
- when MSR_{PR}=1 (user mode), an mtspr or mfspr that specifies an spr value with sprs<sub>s</sub>=0 (user-mode accessible) that represents an unimplemented Special Purpose Register

An Illegal Instruction exception may occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses.

- an instruction that is in invalid form (boundedly undefined results)
an *lswx* instruction for which register RA or register RB is in the range of registers to be loaded (boundedly undefined results)

- a defined instruction that is not implemented by the implementation (*Unimplemented Operation exception*)

**Privileged Instruction exception**

A Privileged Instruction exception occurs when MSRPR=1 and execution is attempted of any of the following kinds of instructions.

- a privileged instruction
- an *mtspr* or *mfspr* instruction that specifies an spr value with spr6=1

**Trap exception**

A Trap exception occurs when any of the conditions specified in a *Trap* instruction are met and the exception is not also enabled as a Debug interrupt. If enabled as a Debug interrupt (i.e. DBCR0TRAP=1, DBCR0IDM=1, and MSR DE=1), then a Debug interrupt will be taken instead of the Program interrupt.

**Unimplemented Operation exception**

An Unimplemented Operation exception may occur when execution is attempted of a defined instruction that is not implemented by the implementation. Otherwise an Illegal Instruction exception occurs.

An Unimplemented Operation exception may also occur when the thread is in 32-bit mode and execution is attempted of an instruction that is part of the 64-Bit category. Otherwise the instruction executes normally.

**SRR0, SRR1, MSR, and ESR are updated as follows:**

- **SRR0**
  - For all Program interrupts except an Enabled exception when in one of the imprecise modes (see Section 4.2.1 on page 1035) or when a disabled exception is subsequently enabled, set to the effective address of the instruction that caused the Program interrupt.
  - For an imprecise Enabled exception, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction has not been executed, and ESRPIE is set to 1. If a subsequent instruction is an *sync* or *isync*, SRR0 will point at the *sync* or *isync* instruction, or at the following instruction.
  - If FPSCR FEX=1 but both MSRFE0=0 and MSRFE1=0, an Enabled exception type Program interrupt will occur imprecisely prior to or at the next synchronizing event if these MSR bits are altered by any instruction that can set the MSR so that the expression
    \[(\text{MSRFE0} \lor \text{MSRFE1}) \& \text{FPSCR}_{\text{FEX}}\]
    is 1. When this occurs, SRR0 is loaded with the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt, and ESRPIE is set to 1.

- **SRR1**
  - Set to the contents of the MSR at the time of the interrupt.

- **MSR**
  - **CM**
    - MSR_{CM} is set to EPCR_{ICM}.
  - **CE, ME, DE**
    - Unchanged
  - **All other defined MSR bits** set to 0.

- **ESR**
  - **PIL**
    - Set to 1 if an Illegal Instruction exception type Program interrupt; otherwise set to 0
  - **PPR**
    - Set to 1 if a Privileged Instruction exception type Program interrupt; otherwise set to 0
  - **PTR**
    - Set to 1 if a Trap exception type Program interrupt; otherwise set to 0
  - **PUO**
    - Set to 1 if an Unimplemented Operation exception type Program interrupt; otherwise set to 0
  - **FP**
    - Set to 1 if the instruction causing the interrupt is a floating-point instruction; otherwise set to 0.
  - **PIE**
    - Set to 1 if a Floating-point Enabled exception type Program interrupt, and the address saved in SRR0 is not the address of the instruction causing the exception (i.e. the instruction that caused FPSCR_{FEX} to be set); otherwise set to 0.
  - **AP**
    - Set to 1 if the instruction causing the interrupt is an Auxiliary Processor instruction; otherwise set to 0.
  - **SPV**
    - Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.
  - **VLEMI**
    - Set to 1 if the instruction causing the interrupt resides in VLE storage.
  - **All other defined ESR bits** set to 0.
  - If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR_{0:51} || 0x0E0. Otherwise, instruction execution resumes at address IVPR_{0:47} || IVOR6_{48:59} || 0b0000.

**7.6.9 Floating-Point Unavailable Interrupt**

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), an attempt is made to execute a floating-point instruction (i.e. any instruction listed in Section 4.6 of Book I), and MSRFP=0.
When a Floating-Point Unavailable interrupt occurs, the hardware suppresses the execution of the instruction causing the Floating-Point Unavailable interrupt.

SRR0, SRR1, and MSR are updated as follows:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: Set to the contents of the MSR at the time of the interrupt.
- **MSR**
  - **CM**: MSR_CM is set to EPCR_ICM.
  - **VLEMI**: Set to 1 if the instruction causing the interrupt resides in VLE storage.
  - **CE, ME, DE**: Unchanged.
  - All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x100. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR748:59|| 0b0000.

### 7.6.10 System Call Interrupt

A System Call interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and a **System Call** (sc) instruction is executed.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, and MSR are updated as follows:

- **SRR0**: Set to the effective address of the instruction after the **sc** instruction.
- **SRR1**: Set to the contents of the MSR at the time of the interrupt.

- **MSR**
  - **CM**: MSR_CM is set to EPCR_ICM.
  - **VLEMI**: Set to 1 if the instruction causing the interrupt resides in VLE storage.
  - **CE, ME, DE**: Unchanged.
  - All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state, MSR0G and GSR0 are set in place of SRR0 and SRR1, respectively. The MSR is set as follows:

- **MSR**
  - **CM**: MSR_CM is set to EPCR_GICM.
  - **CE, ME, GS, DE**: Unchanged.
  - All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x140. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR748:59|| 0b0000.

### 7.6.11 Auxiliary Processor Unavailable Interrupt

An Auxiliary Processor Unavailable interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), an attempt is made to execute an Auxiliary Processor instruction (including Auxiliary Processor loads, stores, and moves), the target Auxiliary Processor is present on the implementation, and the Auxiliary Processor is configured as unavailable. Details of the Auxiliary Processor, its instruction set, and its configuration are implementation-dependent. See User’s Manual for the implementation.

When an Auxiliary Processor Unavailable interrupt occurs, the hardware suppresses the execution of the instruction causing the Auxiliary Processor Unavailable interrupt.

Registers SRR0, SRR1, and MSR are updated as follows:

- **SRR0**: Set to the effective address of the instruction that caused the interrupt.
- **SRR1**: Set to the contents of the MSR at the time of the interrupt.

- **MSR**
  - **CM**: MSR_CM is set to EPCR_ICM.
  - **CE, ME, DE**: Unchanged.
  - All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:31 || 0x140. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR748:59|| 0b0000.

### 7.6.12 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Decrementer exception exists (TSR_DIS=1), and the excep-
tion is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR[DIE]=1 and (MSR[EE]=1 or MSR[GS]=1). Otherwise, the interrupt is enabled by TCR[DIE]=1 and MSR[EE]=1. See Section 9.3 on page 1199.

**Programming Note**

MSR[EE] also enables the External Input and Fixed-Interval Timer interrupts.

SRR0, SRR1, MSR, and TSR are updated as follows:

- **SRR0**: Set to the effective address of the next instruction to be executed.
- **SRR1**: Set to the contents of the MSR at the time of the interrupt.
- **MSR**: 
  - **CM**: MSR[CM] is set to EPCR[CM].
  - **CE, ME, DE**: Unchanged
  - All other defined MSR bits set to 0.
- **TSR** (See Section 9.7.1 on page 1204.):
  - **DIS**: Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address GIVPR[0:51] || 0x160. Otherwise, instruction execution resumes at address GIVPR[0:47] || GIVOR[10:59] || 0b0000.

**7.6.13 Guest Decrementer Interrupt**

A Guest Decrementer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Decrementer exception exists (GTSR[DIS]=1), and the exception is enabled. The interrupt is enabled by GTCR[DIE]=1 and MSR[EE]=1 and MSR[GS]=1. See Section 9.7 on page 1203.

**Programming Note**

Software is responsible for clearing the Decrementer exception status prior to re-enabling the MSR[EE] bit in order to avoid another redundant Decrementer interrupt. To clear the Decrementer exception, the interrupt handling routine must clear TSR[DIS]. Clearing is done by writing a word to TSR using `mtspr` with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Hypervisor software can modify the value of the GTSR by writing the desired value to the GTSRWR. Bits specified in the GTSRWR directly set or clear the corresponding implemented bits in the GTSR.

**7.6.14 Fixed-Interval Timer Interrupt**

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Fixed-Interval Timer exception exists (TSR[FIS]=1), and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR[FIE]=1 and (MSR[EE]=1 or MSR[GS]=1). Otherwise, the interrupt is enabled by TCR[FIE]=1 and MSR[EE]=1. See Section 9.9 on page 1208.
SRR0, SRR1, MSR, and TSR are updated as follows:

**SRR0**
Set to the effective address of the next instruction to be executed.

**SRR1**
Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM**: MSR\textsubscript{CM} is set to EPCR\textsubscript{ICM}.
- CE, ME, DE
  - Unchanged.
- All other defined MSR bits set to 0.

**TSR**
(See Section 9.7.1 on page 1204.)
- **FIS**: Set to 1.

If Interrupt Fixed Offsets [Category: Embedded, Phased-In] are supported, instruction execution resumes at address IVPR\textsubscript{0:51} || 0x180. Otherwise, instruction execution resumes at address IVPR\textsubscript{0:47} || IVOR\textsubscript{1148:59} || 0b0000.

---

**Programming Note**
Software is responsible for clearing the Fixed-Interval Timer exception status prior to re-enabling the MSR\textsubscript{EE} bit in order to avoid another redundant Fixed-Interval Timer interrupt. To clear the Fixed-Interval Timer exception, the interrupt handling routine must clear TSR\textsubscript{FIS}. Clearing is done by writing a word to TSR using \texttt{mtspr} with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Hypervisor software can modify the value of the GTSR by writing the desired value to the GTSRWR. Bits specified in the GTSRWR directly set or clear the corresponding implemented bits in the GTSR.

---

### 7.6.15 Guest Fixed Interval Timer Interrupt

A Guest Decrementer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Decrementer exception exists (GTSR\textsubscript{DIS}=1), and the exception is enabled. The interrupt is enabled by TCR[DIE]=1 and MSR\textsubscript{EE}=1 and MSR\textsubscript{GS}=1. See Section 9.8 on page 1206.

**Programming Note**
MSR\textsubscript{EE} also enables the External Input and Guest Fixed-Interval Timer interrupts.

**GSRR0, GSRR1, MSR, and GTSR** are updated as follows:

**GSRR0**
Set to the effective address of the next instruction to be executed.

---

**Programming Note**
MSR\textsubscript{CE} also enables the Critical Input interrupt.

**GSRR0, GSRR1, MSR, and GTSR** are updated as follows:

**GSRR0**
Set to the effective address of the next instruction to be executed.
CM  MSR_CM is set to EPCR_ICM.
ME  Unchanged.
DE  Unchanged if category E.ED is supported; otherwise set to 0.

All other defined MSR bits set to 0.

TSR  (See Section 9.7.1 on page 1204.)
WIS  Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR_0:51 || 0x1A0. Otherwise, instruction execution resumes at address IVPR_0:47 || IVOR_1248:59 || 0b0000.

7.6.17 Guest Watchdog Timer

Interrupt

A Guest Watchdog Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Guest Watchdog Timer exception exists (GTSR_WIS=1), and the exception is enabled. The interrupt is enabled by GTCR_WIE=1 and MSR_CE = 1 and MSR_GS=1. See Section 9.8 on page 1206.

--- Programming Note

Software is responsible for clearing the Watchdog Timer exception status prior to re-enabling the MSR_CE bit in order to avoid another redundant Watchdog Timer interrupt. To clear the Watchdog Timer exception, the interrupt handling routine must clear TSR_WIS. Clearing is done by writing a word to TSR using mtspwr with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect. Hypervisor software can modify the value of the GTSR by writing the desired value to the GTSRWR. Bits specified in the GTSRWR directly set or clear the corresponding implemented bits in the GTSR.

--- Programming Note

As all Watchdog Timer interrupts are directed to the hypervisor, it is the responsibility of hypervisor software to reflect interrupts generated by the Guest Watchdog Timer to the guest supervisor.

7.6.18 Data TLB Error Interrupt

A Data TLB Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and any of the following Data TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception

Caused when the virtual address associated with a data storage access does not match any valid entry in the TLB as specified in Section 6.7.2 on page 1081.

If a stbcx, sthcx, stwcx, stdcx, or stqcx would not perform its store in the absence of a Data Storage interrupt, and a non-conditional Store to the specified effective address would cause a Data Storage interrupt, it is implementation dependent whether a Data Storage interrupt occurs.

When a Data TLB Error interrupt occurs, the hardware suppresses the execution of the instruction causing the Data TLB Error interrupt.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

SRR0  Set to the effective address of the instruction causing the Data TLB Error interrupt.

Software is responsible for clearing the Guest Watchdog Timer exception status prior to re-enabling the MSR_CE bit in order to avoid another redundant Guest Watchdog Timer interrupt. To clear the Guest Watchdog Timer exception, the interrupt handling routine must clear TSR_WIS. Clearing is done by writing a word to TSR using mtspwr with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Programming Note

Software is responsible for clearing the Guest Watchdog Timer exception status prior to re-enabling the MSR_CE bit in order to avoid another redundant Guest Watchdog Timer interrupt. To clear the Guest Watchdog Timer exception, the interrupt handling routine must clear TSR_WIS. Clearing is done by writing a word to TSR using mtspwr with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.
SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR CM MSR_CM is set to EHSRICM.

CE, ME, DE Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the Storage Access or Cache Management instruction, and within the page whose access caused the Data TLB Error exception.

ESR ST Set to 1 if the instruction causing the interrupt is a Store, dcbi, dcbz, or dcbzep instruction; otherwise set to 0.

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state, GSRR0, GSRR1, GDEAR, and GESR are set in place of SRR0, SRR1, DEAR, and ESR, respectively. The MSR is set as follows:

MSR CM MSR_CM is set to EPCRGICM.

CE, ME, GS, DE Unchanged.

All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following:
- IVPR_{0:47} || IVOR13_{48:59}||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- IVPR_{0:51}||0x1C0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.19 Instruction TLB Error Interrupt

An Instruction TLB Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and any of the following Instruction TLB Error exceptions is presented to the interrupt mechanism.

**TLB Miss exception**

Caused when the virtual address associated with an instruction fetch does not match any valid entry in the TLB as specified in Section 6.7.2 on page 1081.

When an Instruction TLB Error interrupt occurs, the hardware suppresses the execution of the instruction causing the Instruction TLB Miss exception.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Instruction TLB Error interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR CM MSR_CM is set to EPCRICM.

CE, ME, GS, DE Unchanged.

All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following:
- GIVPR_{0:47} || GIVOR13_{48:59}||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- GIVPR_{0:51}||0x1C0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.
GIVPR\textsubscript{0,47} \parallel GIVOR14\textsubscript{48-59} \parallel 0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.

GIVPR\textsubscript{0,51} \parallel 0x1E0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.

IVPR\textsubscript{0,47} \parallel IVOR14\textsubscript{48-59} \parallel 0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.

IVPR\textsubscript{0,51} \parallel 0x1E0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.20 Debug Interrupt

A Debug interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Debug exception exists in the DBSR, and Debug interrupts are enabled (DBCR0\_DM=1 and MSR\_DE=1). A Debug exception occurs when a Debug Event causes a corresponding bit in the DBSR to be set. See Section 10.5.

If the Embedded.Enhanced Debug category is not supported or is supported and is not enabled, CSRR0, CSRR1, MSR, and DBSR are updated as follows. If the Embedded.Enhanced Debug category is supported and is enabled, DSRR0 and DSRR1 are updated as specified below and CSRR0 and CSRR1 are not changed. The means by which the Embedded.Enhanced Debug category is enabled is implementation-dependent.

CSRR0 or DSRR0 [Category: Embedded.Enhanced Debug]

For Debug exceptions that occur while Debug interrupts are enabled (DBCR0\_DM=1 and MSR\_DE=1), CSRR0 is set as follows:

- For Instruction Address Compare (IAC1, IAC2, IAC3, IAC4), Data Address Compare (DAC1R, DAC1W, DAC2R, DAC2W), Trap (TRAP), or Branch Taken (BRT) debug exceptions, set to the address of the instruction causing the Debug interrupt.
- For Instruction Complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the Debug interrupt.
- For Unconditional Debug Event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the Debug interrupt had not occurred.
- For Interrupt Taken (IRPT) debug exceptions, set to the interrupt vector value of the interrupt that caused the Interrupt Taken debug event.
- For Return From Interrupt (RET) debug exceptions, set to the address of the \textit{rfi} instruction that caused the Debug interrupt.
- For Critical Interrupt Taken (CRPT) debug exceptions, DSRR0 is set to the address of the first instruction of the critical interrupt handler. CSRR0 is unaffected.
- For Critical Interrupt Return (CRET) debug exceptions, DSRR0 is set to the address of the \textit{rfci} instruction that caused the Debug interrupt.

See
Section 10.4.10, “Critical Interrupt Return Debug Event [Category: Embedded.Enhanced Debug].”

For Debug exceptions that occur while Debug interrupts are disabled (DBCR0\_IDM=0 or MSR\_DE=0), a Debug interrupt will occur at the next synchronizing event if DBCR0\_IDM and MSR\_DE are modified such that they are both 1 and if the Debug exception Status is still set in the DBSR. When this occurs, CSRR0 or DSRR0 [Category:Embedded.Enhanced Debug] is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the Debug Control Register 0 or MSR and thus caused the interrupt.

**CSRR1** or **DSRR1** [Category: Embedded.Enhanced Debug]
Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM** MSRCM is set to EPCR\_ICM.
- **ME** Unchanged

All other supported MSR bits set to 0.

**DBSR**
Set to indicate type of Debug Event (see Section 10.5.2)

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x040. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR15:59|| 0b0000.

### 7.6.21 SPE/Embedded Floating-Point/Vector Unavailable Interrupt

[Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Vector, Vector]

The SPE/Embedded Floating-Point/Vector Unavailable interrupt occurs when no higher priority exception exists, and an attempt is made to execute an SPE, SPE.Embedded Float Scalar Double, SPE.Embedded Float Vector, or Vector instruction and MSR\_SPV = 0.

When an Embedded Floating-Point Unavailable interrupt occurs, the hardware suppresses the execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

**SRR0** Set to the effective address of the instruction causing the Embedded Floating-Point Unavailable interrupt.

**SRR1** Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM** MSRCM is set to EPCR\_ICM.
- **CE, ME, DE** Unchanged

**ESR**
- **SPV** Set to 1.
- **VLEMI** Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x200. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR32:59|| 0b0000.

**Programming Note**

This interrupt is also used by the Signal Processing Engine in the same manner. It should be used by software to determine if the application is using the upper 32 bits of the GPRs in a 32-bit implementation and thus be required to save and restore them on context switch.
7.6.22 Embedded Floating-Point Data Interrupt
[Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, SPE.Embedded Float Vector]

The Embedded Floating-Point Data interrupt occurs when no higher priority exception exists (see Section 7.9) and an Embedded Floating-Point Data exception is presented to the interrupt mechanism. The Embedded Floating-Point Data exception causing the interrupt is indicated in the SPEFSCR; these exceptions include Embedded Floating-Point Invalid Operation/Input Error (FINV, FINVH), Embedded Floating-Point Divide By Zero (FDBZ, FDBZH), Embedded Floating-Point Overflow (FOV, FOVH), and Embedded Floating-Point Underflow (FUNF, FUNFH).

When an Embedded Floating-Point Data interrupt occurs, the hardware suppresses the execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

- **SRR0**: Set to the effective address of the instruction causing the Embedded Floating-Point Data interrupt.
- **SRR1**: Set to the contents of the MSR at the time of the interrupt.
- **MSR**:
  - CM: MSR_CM is set to EPCRICM.
  - CE, ME, DE: Unchanged
  - VLEMI: Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined MSR bits set to 0.

**ESR**

- SPV: Set to 1.
- VLEMI: Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR_{0,51} || 0x220. Otherwise, instruction execution resumes at address IVPR_{0,47} || IVOR33_{48,59} || 0b0000.

7.6.23 Embedded Floating-Point Round Interrupt
[Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, SPE.Embedded Float Vector]

The Embedded Floating-Point Round interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), SPEFSRCFINXE is set to 1, and any of the following occurs:

- the unrounded result of an Embedded Floating-Point operation is not exact
- an overflow occurs and overflow exceptions are disabled (FOVF or FOVFH is set to 1 and FOVFE is set to 0)
- an underflow occurs and underflow exceptions are disabled (FUNF is set to 1 and FUNFE is set to 0).

The value of SPEFSRCFINXS is 1, indicating that one of the above exceptions has occurred, and additional information about the exception is found in SPEFSRCFGH FG FXH FX.

When an Embedded Floating-Point Round interrupt occurs, the hardware completes the execution of the instruction causing the exception and writes the result to the destination register prior to taking the interrupt.

SRR0, SRR1, MSR, and ESR are updated as follows:

- **SRR0**: Set to the effective address of the instruction following the instruction causing the Embedded Floating-Point Round interrupt.
- **SRR1**: Set to the contents of the MSR at the time of the interrupt.
- **MSR**:
  - CM: MSR_CM is set to EPCRICM.
  - CE, ME, DE: Unchanged
  - VLEMI: Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined MSR bits set to 0.

**ESR**

- SPV: Set to 1.
- VLEMI: Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR_{0,51} || 0x240. Otherwise, instruction execution resumes at address IVPR_{0,47} || IVOR34_{48,59} || 0b0000.
7.6.24 Performance Monitor Interrupt [Category: Embedded.Performance Monitor]

The Performance Monitor interrupt is part of the optional Performance Monitor facility; see Appendix D.

7.6.25 Processor Doorbell Interrupt [Category: Embedded.Processor Control]

A Processor Doorbell Interrupt occurs when no higher priority exception exists, a Processor Doorbell exception is present, and the interrupt is enabled (MSRCE=1). Processor Doorbell exceptions are generated when DBELL messages (see Section 11) are received and accepted by the thread.

If Category: Embedded.Hypervisor is supported, the interrupt is enabled if MSRGS = 1 or MSRCE=1.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.
CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR CM is set to EPCRICM.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x280. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR3648:59|| 0b0000.

7.6.26 Processor Doorbell Critical Interrupt [Category: Embedded.Processor Control]

A Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Processor Doorbell Critical exception is present, and the interrupt is enabled (MSRCE=1). Processor Doorbell Critical exceptions are generated when DBELL_CRIT messages (see Section 11) are received and accepted by the thread.

If Category: Embedded.Hypervisor is supported, the interrupt is enabled if MSRGS = 1 or MSRCE=1.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.
CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR CM is set to EPCRICM.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x2A0. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR3748:59|| 0b0000.

7.6.27 Guest Processor Doorbell Interrupt [Category: Embedded.Hypervisor,Embedded.Processor Control]

A Guest Processor Doorbell Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell exception is present, and the interrupt is enabled (MSRGSE=1). Guest Processor Doorbell exceptions are generated when G_DBELL messages (see Section 11) are received and accepted by the thread.

Programming Note

If an implementation does not support ±Infinity rounding modes and the rounding mode is set to be +Infinity or -Infinity, an Embedded Floating-Point Round interrupt occurs after every Embedded Floating-Point instruction for which rounding might occur regardless of the value of FINXE, provided no higher priority exception exists.

When an Embedded Floating-Point Round interrupt occurs, the unrounded (truncated) result of an inexact high or low element is placed in the target register. If only a single element is inexact, the other exact element is updated with the correctly rounded result, and the FG and FX bits corresponding to the other exact element will both be 0.

The bits FG (FGH) and FX (FXH) are provided so that an interrupt handler can round the result as it desires. FG (FGH) is the value of the bit immediately to the right of the least significant bit of the destination format mantissa from the infinitely precise intermediate calculation before rounding. FX (FXH) is the value of the ‘or’ of all the bits to the right of the FG (FGH) of the destination format mantissa from the infinitely precise intermediate calculation before rounding.

If an implementation does not support ±Infinity rounding modes and the rounding mode is set to be +Infinity or -Infinity, an Embedded Floating-Point Round interrupt occurs after every Embedded Floating-Point instruction for which rounding might occur regardless of the value of FINXE, provided no higher priority exception exists.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x280. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR3648:59|| 0b0000.

CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x280. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR3648:59|| 0b0000.
GSRR0, GSRR1 and MSR are updated as follows:

**GSRR0**
Set to the effective address of the next instruction to be executed.

**GSRR1**
Set to the contents of the MSR at the time of the interrupt.

**MSR**
- **CM**: MSR\(_{CM}\) is set to EPCR\(_{CM}\).
- **CE, ME, DE**: Unchanged.
- All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR\(_{0:51}\) || 0x2C0. Otherwise, instruction execution resumes at address IVPR\(_{0:47}\) || IVOR\(_{38:59}\)|| 0b0000.

--- Programming Note ---

Guest Processor Doorbell interrupts are used by the hypervisor to be notified when the guest operating system has set MSR\(_{EE}\) to 1. This allows the hypervisor to reflect base class interrupts to the guest at a time when the guest is ready to accept them (MSR\(_{GS}=1\) and MSR\(_{EE}=1\)).

--- Programming Note ---

Some guest operating systems running on a hypervisor may use lazy interrupt blocking. That is, when the operating system wants to block interrupts at the interrupt controller, it does not actually perform the blocking operation, but instead sets a value in memory that represents the level at which interrupts are to be blocked. When an actual interrupt occurs, this value is consulted to determine if the interrupt should have been blocked. If so, the current interrupt level that is to be blocked is set in the interrupt controller and the interrupt handling code returns without acknowledging the interrupt. When interrupts are unblocked at a later time, the interrupt will be reasserted by the interrupt controller. When a hypervisor is taking external interrupts and then reflecting them to a guest, the hypervisor must acknowledge the interrupt before reflecting it to the guest since the external interrupt will occur again once MSR\(_{GS}=1\) regardless of the state of MSR\(_{EE}\).

To emulate the behavior required for lazy interrupt blocking by the guest, the hypervisor should execute another `msgsnd` instruction specifying a Guest Processor Doorbell at the time that it is reflecting the interrupt to the guest. When the guest performs its interrupt acknowledge (a hypercall or writing to an interrupt controller register emulated by the hypervisor), the hypervisor can execute a `msgclr` to clear a pending message if there are no other interrupts to be reflected to the guest.
Chapter 7. Interrupts and Exceptions

7.6.28 Guest Processor Doorbell Critical Interrupt [Category: Embedded.Hypervisor, Embedded.Processor Control]

A Guest Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Critical exception is present, and the interrupt is enabled (MSRGS = 1 and MSRCE=1). Guest Processor Doorbell Critical exceptions are generated when G_DBELL_CRIT messages (see Section 11) are received and accepted by the thread.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_CM is set to EPCR_ICM.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0
All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x2E0. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR3948:59|| 0b0000.

Programming Note

Guest Processor Doorbell Critical interrupts are used by the hypervisor to be notified when the guest operating system has set MSRCE to 1. This allows the hypervisor to reflect critical class interrupts to the guest at a time when the guest is ready to accept them (MSRGS=1 and MSRCE=1).

7.6.29 Guest Processor Doorbell Machine Check Interrupt [Category: Embedded.Hypervisor, Embedded.Processor Control]

A Guest Processor Doorbell Machine Check Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Machine Check exception is present, and the interrupt is enabled (MSRGS = 1 and MSRME=1). Guest Processor Doorbell Machine Check exceptions are generated when G_DBELL_MC messages (see Section 11) are received and accepted by the thread.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_CM is set to EPCR_ICM.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0
All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x2E0. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR3948:59|| 0b0000.

Programming Note

Guest Processor Doorbell Machine Check interrupts are used by the hypervisor to be notified when the guest operating system has set MSRME to 1. This allows the hypervisor to reflect machine check class interrupts to the guest at a time when the guest is ready to accept them (MSRGS=1 and MSRME=1).

7.6.30 Embedded Hypervisor System Call Interrupt [Category: Embedded.Hypervisor]

An Embedded Hypervisor System Call interrupt occurs when no higher priority exception exists (see Section 7.9) and a System Call (sc) instruction with LEV = 1 is executed.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction after the sc instruction.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_CM is set to EPCR_ICM.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
CE, ME, DE
Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x300. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR4148:59|| 0b0000.

7.6.31 Embedded Hypervisor Privilege Interrupt [Category: Embedded.Hypervisor]

An Embedded Hypervisor Privilege interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an Embedded Hypervisor Privilege exception is presented to the exception mechanism.

An Embedded Hypervisor Privilege exception occurs when MSRGS = 1 and MSRPR = 0 and execution is attempted of any of the following:

- a hypervisor-privileged instruction
- an mispr or mispr instruction that specifies an SPR that is hypervisor privileged
- a tlbwe instruction and Category: Embedded.Hypervisor.LRAT is not implemented.
- a tlbwe, tlbsrx, or tlibx instruction and EPCR DGTM=1
- a tlbwe instruction that attempts to write a TLB entry for which TLBV=1 and TLBIPROT=1 when MAS0WO=0b00
- a tlbwe instruction that attempts to write a TLB entry when MAS1V=1, MAS1IPROT=1, and MAS0WO=0b00
- a mtpr or mtpr instruction and MSRPPMMP = 1
- a Cache Locking instruction and MSRUClep = 1

An Embedded Hypervisor Privilege exception may occur for the following implementation dependent reasons when MSRGS = 1 and MSRPR = 0 and execution is attempted of any of the following:

- a tlbwe instruction that attempts to write a TLB entry for which TLBV=0 and TLBIPROT=1 when MAS0WO=0b00
- a tlbwe instruction that attempts to write a TLB entry when MAS1V=0, MAS1IPROT=1, and MAS0WO=0b00
- a tlbwe instruction that attempts to write a TLB entry for which TLBIPROT=1 when MAS0WO=0b01
- a tlbwe instruction that attempts to write a TLB entry when MAS1IPROT=1, and MAS0WO=0b01
- a tlbwe instruction that attempts to write a TLB entry to an array that is disallowed by the implementation
- an implementation dependent instruction or SPR which is hypervisor privileged

An Embedded Hypervisor Privilege exception also occurs when execution is attempted of an ehpriv instruction, regardless of the state of the thread.

Execution of the instruction causing the interrupt is suppressed and SRR0, SRR1, and MSR are updated as follows:

**SRR0** Set to the effective address of the instruction causing the Embedded Hypervisor Privilege interrupt.

**SRR1** Set to the contents of the MSR at the time of the interrupt.

**MSR**
- CM MSR CM is set to EPCR ICMM.
- VLEM Set to 1 if the instruction causing the interrupt resides in VLE storage.
- CE, ME, DE Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address IVPR0:51 || 0x320. Otherwise, instruction execution resumes at address IVPR0:47 || IVOR4148:59|| 0b0000.

7.6.32 LRAT Error Interrupt [Category: Embedded.Hypervisor.LRAT]

An LRAT Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an LRAT Miss exception is presented to the interrupt mechanism.

An LRAT Miss exception is caused by either of the following:

- A tlbwe instruction is executed in guest supervisor state and the logical page number (RPN specified by MAS7 and MAS3 and page size specified by MAS1TSIZE) does not match any valid entry in the LRAT.
- A Page Table translation is performed and the associated PTEARPN is to be treated as an LPN (the Embedded.Hypervisor category is supported) and the logical page number (RPN based on PTEARPN and page size specified by PTEPS) does not match any valid entry in the LRAT.

When an LRAT Error interrupt occurs, the hardware suppresses the execution of the instruction causing the LRAT Error interrupt.

**SRR0, SRR1, MSR, ESR, and LPER** are updated as follows:

**SRR0** Set to the effective address of the instruction causing the LRAT Error interrupt.

**SRR1** Set to the contents of the MSR at the time of the interrupt.
MSR

CMMSR_{CM} is set to EPCR_{ICM}.
CE, ME, DE Unchanged.

All other defined MSR bits are set to 0.

DEAR

If the LRAT Error interrupt occurred for a Page Table translation, set to the effective address of a byte that is both within the range of the bytes being accessed by the Storage Access or Cache Management instruction, and within the page whose access caused the LRAT Miss exception. Otherwise, undefined.

ESR

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.
ST Set to 1 if the instruction causing the interrupt is a Store or 'store-class' Cache Management instruction and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.
AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.
SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation, the instruction is a Load or Store, and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.
DATA Set to 1 if the interrupt is due to an LRAT miss resulting from a Page Table translation of a Load, Store or Cache Management operand address; otherwise set to 0.
TLBI Set to 1 if a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt; otherwise set to 0.
PT Set to 1 if the cause of the interrupt is an LRAT miss exception on a Page Table translation. Set to 0 if the cause of the interrupt is an LRAT miss exception on a tlbwe.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage, the instruction is a Load, Store, or Cache Management instruction, and the translation of the operand address causes the LRAT Miss exception.
EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction, the instruction is a Load, Store, or Cache Management instruction, and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.

All other defined ESR bits are set to 0.

LPER

Set to the values of the ARPN and PS fields from the PTE that was used to translate a virtual address for an instruction fetch, Load, Store or Cache Management instruction that caused an LRAT Error interrupt as a result of an LRAT Miss exception.

If Interrupt Fixed Offsets [Category: Embedded,Phased-In] are supported, instruction execution resumes at address IVPR_{0:47} || 0x0340. Otherwise, instruction execution resumes at address IVPR_{0:47} || IVOR42_{48:59} || 0b0000.
7.7 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. Unaligned Load and Store instructions, or Load Multiple, Store Multiple, Load String, and Store String instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. In order to guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as Guarded, and must use an elementary (non-string or non-multiple) load or store that is aligned on an operand-sized boundary.

In order to guarantee that Load and Store instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

- For an elementary Load, no part of the target register RT or FRT, will have been altered.
- For ‘with update’ forms of Load or Store, the update register, register RA, will not have been altered.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

- For any Store, some of the bytes at the target storage location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for Store Conditional instructions, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared.
- For any Load, some of the bytes at the addressed storage location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).
- For Load Multiple or Load String, some of the registers in the range to be loaded may have been altered. Including the addressing registers (RA, and possibly RB) in the range to be loaded is a programming error, and thus the rules for partial execution do not protect against overwriting of these registers.

In no case will access control be violated.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary, aligned, guarded loads and stores. All others may be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any Load or Store (except elementary, aligned, guarded):
   - Any asynchronous interrupt
   - Machine Check
   - Program (Imprecise Mode Floating-Point Enabled)
   - Program (Imprecise Mode Auxiliary Processor Enabled)

2. Unaligned elementary Load or Store, or any multiple or string:
   - All of the above listed under item 1, plus the following:
     - Data Storage (if the access crosses a protection boundary)
     - Debug (Data Address Compare)

3. mtcrf may also be partially executed due to the occurrence of any of the interrupts listed under item 1 at the time the mtcrf was executing.
   - All instructions prior to the mtcrf have completed execution. (Some storage accesses generated by these preceding instructions may not have completed.)
   - No subsequent instruction has begun execution.
   - The mtcrf instruction (the address of which was saved in SRR0/CSRR0/MCSR0/DSRR0 [Category: Embedded.Enhanced Debug] at the occurrence of the interrupt), may appear not to have begun or may have partially executed.
7.8 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which could cause the generation of an interrupt. Furthermore, for interrupts classes other than the Machine Check interrupt and critical interrupts, the architecture does not provide for reporting more than one interrupt of the same class (unless the Embedded.Enhanced Debug category is supported). Therefore, the architecture defines that interrupts are ordered with respect to each other, and provides a masking mechanism for certain persistent interrupt types.

When an interrupt is masked (disabled), and an event causes an exception that would normally generate the interrupt, the exception persists as a status bit in a register (which register depends upon the exception type). However, no interrupt is generated. Later, if the interrupt is enabled (unmasked), and the exception status has not been cleared by software, the interrupt due to the original exception event will then finally be generated.

All asynchronous interrupts can be masked. In addition, certain synchronous interrupts can be masked. An example of such an interrupt is the Floating-Point Enabled exception type Program interrupt. The execution of a floating-point instruction that causes the FPCR_FE bit to be set to 1 is considered an exception event, regardless of the setting of MSR_FE0,FE1. If MSR_FE0,FE1 are both 0, then the Floating-Point Enabled exception type of Program interrupt is masked, but the exception persists in the FPCR_FE bit. Later, if the MSR_FE0,FE1 bits are enabled, the interrupt will finally be generated.

The architecture enables implementations to avoid situations in which an interrupt would cause the state information (saved in Save/restore Registers) from a previous interrupt to be overwritten and lost. In order to do this, the architecture defines interrupt classes in a hierarchical manner. At each interrupt class, hardware automatically disables any further interrupts associated with the interrupt class by masking the interrupt enable in the MSR when the interrupt is taken. In addition, each interrupt class masks the interrupt enable in the MSR for each lower class in the hierarchy. The hierarchy of interrupt classes is as follows from highest to lowest:

<table>
<thead>
<tr>
<th>Interrupt Class</th>
<th>MSR Enables Cleared</th>
<th>Save/Restore Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Check</td>
<td>ME,DE,CE,EE</td>
<td>MSRRO/1</td>
</tr>
<tr>
<td>Debug¹</td>
<td>DE,CE,EE</td>
<td>DSRR0/1</td>
</tr>
<tr>
<td>Critical</td>
<td>CE,EE</td>
<td>CSRR0/1</td>
</tr>
<tr>
<td>Base</td>
<td>EE</td>
<td>SRR0/1</td>
</tr>
<tr>
<td>Guest &lt;E,HV&gt;</td>
<td>EE</td>
<td>GSRR0/1</td>
</tr>
</tbody>
</table>

¹ The Debug interrupt class is Category: E.ED.

Note: MSR_DS may be cleared when a critical interrupt occurs if Category: E.ED is not supported.

### Figure 71. Interrupt Hierarchy

[Category: Embedded.Hypervisor]

The masking of interrupts is affected by MSR_GS and whether the interrupt is directed to the guest supervisor state or the hypervisor state. In general, interrupts directed to the hypervisor state (with the exception of Guest Processor Doorbell type interrupts), are enabled if MSR_GS = 1 regardless of the value of other MSR enables. Interrupts directed to the guest supervisor state are enabled if the associated MSR enables are set and MSR_GS = 1.

If the Embedded.Enhanced Debug category is not supported (or is supported and is not enabled), then the Debug interrupt becomes a Critical class interrupt and all critical class interrupts will clear DE, CE, and EE in the MSR.

Base Class interrupts that occur as a result of precise exceptions are not masked by the EE bit in the MSR and any such exception that occurs prior to software saving the state of SRR0/1 in a base class exception handler will result in a situation that could result in the loss of state information.

This first step of the hardware clearing the MSR enable bits lower in the hierarchy shown in Figure 71 prevents any subsequent asynchronous interrupts from overwriting the Save/Restore Registers (SRR0/SRR1, CSR0/CSRR1, MCSRR0/MCSRR1, or DSRR0/DSRR1 [Category: Embedded,Enhanced Debug]), prior to software being able to save their contents. Hardware also automatically clears, on any interrupt, MSR_PP,FP,FE0,FE1,IS,DS. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, guaranteeing that interrupt classes lower in the hierarchy do not occur and thus do not overwrite the Save/Restore Registers (SRR0/SRR1, CSR0/CSRR1, DSRR0/DSRR1 [Category: Embedded,Enhanced Debug], or MCSRR0/MCSRR1) also requires the cooperation of system software. Specifically, system software must avoid the exe-
cution of instructions that could cause (or enable) a subsequent interrupt, if the contents of the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], or MCSRR0/MCSRR1) have not yet been saved.

7.8.1 Guidelines for System Software

The following list identifies the actions that system software must avoid, prior to having saved the Save/Restore Registers' contents:

- Re-enabling an interrupt class that is at the same or a lower level in the interrupt hierarchy. This includes the following actions:
  - Re-enabling of MSR_{EE}
  - Re-enabling of MSR_{CE,EE} in critical class interrupt handlers, and if the Embedded.Enhanced Debug category is not supported, re-enabling of MSR_{DE}.
  - [Category: Embedded.Enhanced Debug] Re-enabling of MSR_{CE,EE,DE} in Debug class interrupt handlers
  - Re-enabling of MSR_{EE,CE,DE,ME} in Machine Check interrupt handlers.

- Branching (or sequential execution) to addresses for which any of the following conditions are true.
  - The address is not mapped by the TLB or the Page Table or is mapped without UX=1 or SX=1 permission.
  - Both the Embedded.Hypervisor LRAT and the Embedded Page.Table category are supported, MSR_{GS}=1, and the effective address is mapped by the Page Table but the LPN is not mapped by the LRAT.

This prevents Instruction Storage, LRAT Error, and Instruction TLB Error interrupts.

- Load, Store or Cache Management instructions to addresses for which any of the following conditions are true.
  - The address is not mapped by the TLB or the Page Table or is mapped without the required access permissions.
  - Both the Embedded.Hypervisor LRAT and the Embedded Page.Table category are supported, MSR_{GS}=1, and the effective address is mapped by the Page Table but the LPN is not mapped by the LRAT.

This prevents Data Storage, LRAT Error, and Data TLB Error interrupts.

- Execution of any floating-point instruction
  - This prevents Floating-Point Unavailable interrupts. Note that this interrupt would occur upon the execution of any floating-point instruction, due to the automatic clearing of MSR_{FP}. However, even if software were to re-enable MSR_{FP}, floating-point instructions must still be avoided in order to prevent Program interrupts due to various possible Program interrupt exceptions (Floating-Point Enabled, Unimplemented Operation).

- Re-enabling of MSR_{PR}
  - This prevents Privileged Instruction exception type Program interrupts. Alternatively, software could re-enable MSR_{PR}, but avoid the execution of any privileged instructions.

- Execution of any Auxiliary Processor instruction
  - This prevents Auxiliary Processor Unavailable interrupts, and Auxiliary Processor Enabled type and Unimplemented Operation type Program interrupts.

- Execution of any Illegal instructions
  - This prevents Illegal Instruction exception type Program interrupts.

- Execution of any instruction that could cause an Alignment interrupt
  - This prevents Alignment interrupts. Included in this category are any string or multiple instructions, and any unaligned elementary load or store instructions. See Section 7.6.7 on page 1170 for a complete list of instructions that may cause Alignment interrupts.

It is not necessary for hardware or software to avoid interrupts higher in the interrupt hierarchy (see Figure 71 on page 1187) from within interrupt handlers (and hence, for example, hardware does not automatically clear MSR_{CE,ME,DE} upon a base class interrupt), since interrupts at each level of the hierarchy use different pairs of Save/Restore Registers to save the instruction address and MSR (i.e. SRR0/SRR1 for base class interrupts, and MCSRR0/MCSRR1,DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], or CSR0/CSR1 for non-base class interrupts). The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of interrupts lower in the hierarchy from occurring within interrupt handlers, even though the these interrupts use different Save/Restore Register pairs. This is because the interrupt higher in the hierarchy may have occurred from within a interrupt handler for an interrupt lower in the hierarchy prior to the interrupt handler having saved the Save/Restore Registers. Therefore, within an interrupt handler, Save/Restore Registers for all interrupts lower in the hierarchy may contain data that is necessary to the system software.
7.8.2 Interrupt Order

The following is a prioritized listing of the various enabled interrupts for which exceptions might exist simultaneously:

1. Synchronous (Non-Debug) Interrupts:
   - Data Storage
   - Instruction Storage
   - Alignment
   - Program
   - Embedded Hypervisor Privilege [Category: Embedded.Hypervisor]
   - Floating-Point Unit Unavailable
   - Auxilliary Processor Unavailable
   - Embedded Floating-Point Unavailable [Category: SP.Embedded Float_*]
   - SPE/Embedded Floating-Point/Vector Unavailable [Category: SP.Embedded Float_*]
   - Embedded Floating-Point Data [Category: SP.Embedded Float_*]
   - Embedded Floating-Point Round [Category: SP.Embedded Float_*]
   - System Call
   - Embedded Hypervisor System Call [Category: Embedded.Hypervisor]
   - Data TLB Error
   - Instruction TLB Error
   - LRAT Error [Category: Embedded.Hypervisor.LRAT]

Only one of the above types of synchronous interrupts may have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see Section 7.9 on page 1190) and the requirements of the Sequential Execution Model.

2. Machine Check


4. Debug

5. Critical Input

6. Watchdog Timer

7. Guest Watchdog Timer [Category: Embedded.Hypervisor]

8. Processor Doorbell Critical [Category: Embedded.Processor Control]


10. External Input Category: Embedded.Processor Control

11. Fixed-Interval Timer Category: Embedded.Processor Control


13. Decrementer Category: Embedded.Processor Control


15. Processor Doorbell [Category: Embedded.Processor Control]


17. Embedded Performance Monitor

Even though, as indicated above, the base, synchronous exception types listed under item 1 are generated with higher priority than the non-base interrupt classes listed in items 2-6, the fact is that these base class interrupts will immediately be followed by the highest priority existing interrupt in items 2-5, without executing any instructions at the base class interrupt handler. This is because the base interrupt classes do not automatically disable the MSR mask bits for the interrupts listed in 2-5. In all other cases, a particular interrupt class from the above list will automatically disable any subsequent interrupts of the same class, as well as all other interrupt classes that are listed below it in the priority order.
7.9 Exception Priorities

All synchronous (precise and imprecise) interrupts are reported in program order, as required by the Sequential Execution Model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt will then be generated with all of those exception types reported cumulatively, in both the ESR, and any status registers associated with the particular exception type (e.g. the Floating-Point Status and Control Register).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a single enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time, there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 7.8.2 on page 1189. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

Because unaligned Load and Store instructions, or Load Multiple, Store Multiple, Load String, and Store String instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. The exception priority mechanism applies to each of the multiple storage accesses in the order they are performed by the implementation.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled will have no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it will prevent the setting of that other exception, independent of whether that other exception’s corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type will be permitted to be generated at any given time. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

### Programming Note

Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In these cases, the exceptions are listed in the order suggested by the sequential execution model.

#### 7.9.1 Exception Priorities for Defined Instructions

#### 7.9.1.1 Exception Priorities for Defined Floating-Point Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined Floating-Point Load and Store instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error for instruction fetch [Categories: E.PT and E.HVLRT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
7. Floating-Point Unavailable
8. Program (Unimplemented Operation)
9. Data TLB Error
10. Data Storage (all types)
11. Alignment
12. LRAT Error for data access [Categories: E.PT and E.HVLRT]
13. Debug (Data Address Compare)
14. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare), and is not causing any of the exceptions listed in items 2-11, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

#### 7.9.1.2 Exception Priorities for Other Defined Load and Store Instructions and Defined Cache Management Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any other defined Load or Store instruction, or defined Cache Management instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error for instruction fetch [Categories: E.PT and E.HVLRT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
Chapter 7. Interrupts and Exceptions

7. Program (Unimplemented Operation)
8. Embedded Hypervisor Privilege [Category: E.HV]
9. Data TLB Error
10. Data Storage (all types)
11. Alignment
12. LRAT Error for data access [Categories: E.PT and E.HV.LRAT]
13. Debug (Data Address Compare)
14. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare), and is not causing any of the exceptions listed in items 2-11, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.3 Exception Priorities for Other Defined Floating-Point Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined floating-point instruction other than a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Floating-Point Unavailable
7. Program (Unimplemented Operation)
8. Program (Floating-point Enabled)
9. Debug (Instruction Complete)

7.9.1.4 Exception Priorities for Defined Privileged Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined privileged instruction, except dcbi, rfi, and rfc1 instructions.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT] (for hardware tablewalk)
5. Program (Illegal Instruction, except for TLB management instructions with invalid MAS settings, see 9)
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Embedded Hypervisor Privilege [Category: E.HV]
9. Program (Illegal Instruction, special case for TLB management instructions with invalid MAS settings)
10. LRAT Error [Category: E.HV.LRAT] (for tlbwe)
11. Debug (Instruction Complete)

For mtmsr, mtspr (DBCR0, DBCR1, DBCR2), mtsspr (TCR), and mtsspr (TSR), if they are not causing Debug (Instruction Address Compare) nor Program (Privileged Instruction) exceptions, it is possible that they are simultaneously enabling (via mask bits) multiple existing exceptions (and at the same time possibly causing a Debug (Instruction Complete) exception). When this occurs, the interrupts will be handled in the order defined by Section 7.8.2 on page 1189.

7.9.1.5 Exception Priorities for Defined Trap Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined Trap instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Unimplemented Operation)
7. Debug (Trap)
8. Program (Trap)
9. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Trap), and is not causing any of the exceptions listed in items 2-6, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.6 Exception Priorities for Defined System Call Instruction

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined System Call instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Unimplemented Operation)
7. System Call
8. Embedded Hypervisor System Call [Category: E.HV]
9. Debug (Instruction Complete)

7.9.1.7 Exception Priorities for Defined Branch Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined branch instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Unimplemented Operation)
7. Debug (Branch Taken)
8. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Branch Taken), and is not causing any of the exceptions listed in items 2-6, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.8 Exception Priorities for Defined Return From Interrupt Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of an rfi, rfci, rfmci, rfdi [Category: Embedded,Enhanced Debug], rfgi [Category: Embedded,Hypervisor] instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Debug (Return From Interrupt)
9. Debug (Instruction Complete)

If the rfi or rfci, rfmci, or rfdi [Category: Embedded,Enhanced Debug] or rfgi [Category: Embedded,Hypervisor] instruction is causing both a Debug (Instruction Address Compare) and a Debug (Return From Interrupt), and is not causing any of the exceptions listed in items 2-6, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.9 Exception Priorities for Other Defined Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of all other instructions not listed above.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error instruction fetch [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Embedded Hypervisor Privilege <E.HV>
9. LRAT Error for data access for tlbwe [Category: E.HV.LRAT]
10. Debug (Instruction Complete)

7.9.2 Exception Priorities for Reserved Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any reserved instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)
Chapter 8. Reset and Initialization

8.1 Background

This chapter describes the requirements for thread reset. This includes both the means of causing reset, and the specific initialization that is required to be performed automatically by the hardware. This chapter also provides an overview of the operations that should be performed by initialization software.

In general, the specific actions taken by a thread upon reset are implementation-dependent. Also, it is the responsibility of system initialization software to initialize the majority of thread and system resources after reset. Implementations are required to provide a minimum thread initialization such that this system software may be fetched and executed, thereby accomplishing the rest of system initialization.

8.2 Reset Mechanisms

This specification defines two mechanisms for internally invoking a thread reset operation using either the Watchdog Timer (see Section 9.11 on page 1208) or the Debug facilities using DBCR0RST (see Section 10.5.1.1 on page 1221). In addition, implementations will typically provide additional means for invoking a reset operation, using an external mechanism such as a signal pin, which when activated, causes the thread to be reset.

8.3 Thread State after Reset

The initial thread state is controlled by the register contents after reset. In general, the contents of most registers are undefined after reset.

The hardware is only guaranteed to initialize those registers (or specific bits in registers) which must be initialized in order for software to be able to reliably perform the rest of system initialization.

The Thread Enable Register, Machine State Register, Processor Version Register, and a TLB entry are updated as follows.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Setting</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>0</td>
<td>Computation Mode (set to 32-bit mode)</td>
</tr>
<tr>
<td>GS</td>
<td>0</td>
<td>Hypervisor state &lt;E.HV&gt;</td>
</tr>
<tr>
<td>UCLE</td>
<td>0</td>
<td>User Cache Locking Enable</td>
</tr>
<tr>
<td>SPV</td>
<td>0</td>
<td>SPE/Embedded Floating-Point/Vector Unavailable</td>
</tr>
<tr>
<td>CE</td>
<td>0</td>
<td>Critical Input interrupts disabled</td>
</tr>
<tr>
<td>DE</td>
<td>0</td>
<td>Debug interrupts disabled</td>
</tr>
<tr>
<td>EE</td>
<td>0</td>
<td>External Input interrupts disabled</td>
</tr>
<tr>
<td>PR</td>
<td>0</td>
<td>Supervisor mode</td>
</tr>
<tr>
<td>FP</td>
<td>0</td>
<td>FP unavailable</td>
</tr>
<tr>
<td>ME</td>
<td>0</td>
<td>Machine Check interrupts disabled</td>
</tr>
<tr>
<td>FE0</td>
<td>0</td>
<td>FP exception type Program interrupts disabled</td>
</tr>
<tr>
<td>FE1</td>
<td>0</td>
<td>FP exception type Program interrupts disabled</td>
</tr>
<tr>
<td>IS</td>
<td>0</td>
<td>Instruction Address Space 0</td>
</tr>
<tr>
<td>DS</td>
<td>0</td>
<td>Data Address Space 0</td>
</tr>
<tr>
<td>PMM</td>
<td>0</td>
<td>Performance Monitor Mark</td>
</tr>
</tbody>
</table>

Figure 72. Machine State Register Initial Values

Logical Partition Identification Register [Category: Embedded.Hypervisor]

The Logical Partition Identification Register (LPIDR) is set to 0.

Thread Enable Register [Category: Embedded Multi-Threading]

The TEN is set to the value 0x0000_0000_0000_0001, indicating that only thread 0 is enabled.
Processor Version Register

Implementation-Dependent. (This register is read-only, and contains a value which identifies the specific implementation)

**TLB entry**

A TLB entry (which entry is implementation-dependent) is initialized in an implementation-dependent manner that maps the last page in the implemented effective storage address space, with the following field settings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Setting</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>1</td>
<td>valid</td>
</tr>
<tr>
<td>EPN</td>
<td>see below</td>
<td>Represents the last page in effective address space</td>
</tr>
<tr>
<td>RPN</td>
<td>see below</td>
<td>Represents the last page in physical address space</td>
</tr>
<tr>
<td>TS</td>
<td>0</td>
<td>translation address space 0</td>
</tr>
<tr>
<td>IND</td>
<td>0</td>
<td>direct entry</td>
</tr>
<tr>
<td>TLPID</td>
<td>0</td>
<td>translation logical partition ID</td>
</tr>
<tr>
<td>TGS</td>
<td>0</td>
<td>translation hypervisor state</td>
</tr>
<tr>
<td>SIZE</td>
<td>?</td>
<td>smallest page size supported</td>
</tr>
<tr>
<td>W</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>I</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>M</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>G</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>E</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>U0</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>U1</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>U2</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>U3</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>TID</td>
<td>?</td>
<td>implementation-dependent value, but page must be accessible</td>
</tr>
<tr>
<td>UX</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>UR</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>UW</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>SX</td>
<td>1</td>
<td>page is execute accessible in supervisor mode</td>
</tr>
<tr>
<td>SR</td>
<td>1</td>
<td>page is read accessible in supervisor mode</td>
</tr>
<tr>
<td>SW</td>
<td>1</td>
<td>page is write accessible in supervisor mode</td>
</tr>
<tr>
<td>VLE</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>ACM</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>IPR</td>
<td>?</td>
<td>implementation-dependent value</td>
</tr>
<tr>
<td>VF</td>
<td>0</td>
<td>no virtualization fault</td>
</tr>
</tbody>
</table>

**Figure 73. TLB Initial Values**

The initial settings of EPN and RPN are dependent upon the number of bits implemented in the EPN and RPN fields and the minimum page size supported by the implementation. For example, an implementation that supports 64KB pages as the smallest size and 32 bits of effective address would implement a 16 bit EPN and set the initial value of the EPN field of the TLB boot entry to 2^{16}.1 (0xFFFF) while an implementation that supports 4K pages as the smallest size and 32 bits of effective address would implement a 20 bit EPN and
set the initial value of the boot entry to \(2^{20} - 1\) (0xFFFFF).

Instruction execution begins at the last word address of the page mapped by the boot TLB entry. Note that this address is different from the System Reset interrupt vector specified in Book III-S.

An implementation may provide additional methods for initializing the TLB entry used for initial boot by providing an implementation-dependent RPN, or initializing other TLB entries.

If Category: Embedded Multi-threading.Thread Management is not supported, instruction execution for other threads begins at the last word address of the effective address space; otherwise execution begins at the address specified by the NIA register corresponding to the thread.

### 8.4 Software Initialization Requirements

When reset occurs, the thread is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the thread and system configuration. The initialization code described in this section is the minimum recommended for configuring the thread to run application code.

Initialization code should configure the following resources:

- Invalidate the instruction cache and data cache (implementation-dependent).
- Initialize system memory as required by the operating system or application code.
- Initialize the Interrupt Vector Prefix Register and Interrupt Vector Offset Register.
- Initialize other registers as needed by the system.
- Initialize off-chip system facilities.
- Dispatch the operating system or application code.
Chapter 9. Timer Facilities

9.1 Overview

The Time Base, Decrementer, Fixed-interval Timer, and Watchdog Timer provide timing functions for the system. The remainder of this section describes these registers and related facilities.

9.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 74) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.

Figure 74. Time Base

The Time Base bits 0:59 increment until their value becomes 0xFFF_FFFF_FFFF_FFFF (2^{59} - 1), at the next increment their value becomes 0x000_0000_0000_0000. There is no interrupt or other indication when this occurs.

Time base bits 60:63 may increment at a variable rate. When the value of bit 59 changes, bits 60:63 are set to zero; if bits 60:63 increment to 0xF before the value of bit 59 changes, they remain at 0xF until the value of bit 59 changes.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

\[
T_{TB} = \frac{2^{64} \times 32}{1 \text{ GHz}} = 5.90 \times 10^{11} \text{ seconds}
\]

which is approximately 18,700 years.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.

2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Power ISA system. The Time Base update frequency is not required to be constant. What is required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base bits 0:59 changes, and a means to determine what the current update frequency is.

- The update frequency of the Time Base bits 0:59 is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in user mode (MSR_{PR}=1). If the means is under software control, it must be privileged. There must be a method for getting all Time Bases in the system to start incrementing with values that are identical or almost identical.
9.2.1 Writing the Time Base

Writing the Time Base is hypervisor privileged. Reading the Time Base is not privileged, it is discussed in Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The `mttbl` and `mttbu` extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the `mtspr` instruction; see Appendix B, “Assembler Extended Mnemonics” on page 1245.

The Time Base can be written by a sequence such as:

```
   lwz Rx,upper # load 64-bit value for
   lwz Ry,lower # TB into Rx and Ry
   li  Rz,0
   mttbl Rz  # set TBL to 0
   mttbu Rx  # set TBU
   mttbl Ry  # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

---

**Programming Note**

If software initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from $2^{64}-1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

See the description of the Time Base in Book II, for ways to compute time of day in POSIX format from the Time Base.

---

**Virtualized Implementation Note**

In virtualized implementations, TBU and TBL are read-only.

---

**Programming Note**

The instructions for writing the Time Base are mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.
9.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

Figure 75. Decrementer

Decrementer bits 32:59 count down until their value becomes 0x0000_0000, at the next increment their value becomes 0xFFF_FFFF. Decrementer bits 60:63 may decrement at a variable rate. When the value of bit 59 changes, bits 60:63 are set to 0xF; if bits 60:63 decrement to 0x0 before the value of bit 59 changes, they remain at 0x0 until the value of bit 59 changes.

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 9.2), and if the Time Base update frequency is constant, the period would be

\[ T_{\text{DEC}} = \frac{2^{32} \times 32}{1 \text{ GHz}} = 137 \text{ seconds}. \]

The Decrementer counts down.

The operation of the Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
3. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

9.3.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the `mfspr` and `mtspr` instructions, both of which are hypervisor privileged. When `mfspr` and `mtspr` are executed in guest supervisor state, the access to the DEC is mapped to the GDEC. Using an extended mnemonic (see Appendix B, “Assembler Extended Mnemonics” on page 1245), the Decrementer can be written from GPR Rx using:

`mtdec Rx`

The Decrementer can be read into GPR Rx using:

`mfdec Rx`

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

9.3.2 Decrementer Events

A Decrementer event occurs when a decrement occurs on a Decrementer value of 0x0000_0001.

Upon the occurrence of a Decrementer event, the Decrementer may be reloaded from a 32-bit Decrementer Auto-Reload Register (DECAR). See Section 9.5. Upon the occurrence of a Decrementer event, the Decrementer has the following basic modes of operation.

Decrement to one and stop on zero

If TCR\_ARE=0, TSR\_DIS is set to 1, the value 0x0000_0000 is then placed into the DEC, and the Decrementer stops decrementing.

A Decrementer interrupt occurs when no higher priority interrupt exists, a Decrementer exception exists, and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR\_DIE=1 and (MSR\_EE=1 or MSR\_GS=1). Otherwise, the interrupt is enabled by TCR\_DIE=1 and MSR\_EE=1. See Section 7.6.12, “Decrementer Interrupt” on page 1173 for details of register behavior caused by the Decrementer interrupt.

Decrement to one and auto-reload

If TCR\_ARE=1, TSR\_DIS is set to 1, the contents of the Decrementer Auto-Reload Register is then placed into the DEC, and the Decrementer continues decrementing from the reloaded value.

A Decrementer interrupt occurs when no higher priority interrupt exists, a Decrementer exception exists, and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR\_DIE=1 and (MSR\_EE=1 or

---

**Programming Note**

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.
MSR_DS=1). Otherwise, the interrupt is enabled by TCR_DSIE=1 and MSR_DSIE=1. See Section 7.6.12, "Decrementer Interrupt" on page 1173 for details of register behavior caused by the Decrementer interrupt.

Forcing the Decrementer to 0 using the \texttt{mtspr} instruction will not cause a Decrementer exception; however, decrementing which was in progress at the instant of the \texttt{mtspr} may cause the exception. To eliminate the Decrementer as a source of exceptions, set TCR_DSIE to 0 (clear the Decrementer Interrupt Enable bit).

If it is desired to eliminate all Decrementer activity, the procedure is as follows:

1. Write 0 to TCR_DSIE. This will prevent Decrementer activity from causing exceptions.
2. Write 0 to TCR_ARE to disable the Decrementer auto-reload.
3. Write 0 to Decrementer. This will halt Decrementer decrementing. While this action will not cause a Decrementer exception to be set in TSRDIS, a near simultaneous decrement may have done so.
4. Write 1 to TSRDIS. This action will clear TSRDIS to 0 (see Section 9.7.1 on page 1204). This will clear any Decrementer exception which may be pending. Because the Decrementer is frozen at zero, no further Decrementer events are possible.

If the auto-reload feature is disabled (TCR_ARE=0), then once the Decrementer decrements to zero, it will stay there until software reloads it using the \texttt{mtspr} instruction.

On reset, TCR_ARE is set to 0. This disables the auto-reload feature.

9.4 Guest Decrementer [Category: Embedded.Hypervisor]

The Guest Decrementer (GDEC) is a 32-bit decrementing counter that provides a mechanism for causing a Guest Decrementer interrupt after a programmable delay. The contents of the Guest Decrementer are treated as a signed integer.

![Figure 76. Guest Decrementer](image)

Guest Decrementer bits 32:59 count down until their value becomes 0x000_0000; at the next increment their value becomes 0xFFFF_FFFF. Guest Decrementer bits 60:63 may decrement at a variable rate. When the value of bit 59 changes, bits 60:63 are set to 0xF; if bits 60:63 decrement to 0x0 before the value of bit 59 changes, they remain at 0x0 until the value of bit 59 changes.

The Guest Decrementer is driven by the same frequency as the Time Base. The period of the Guest Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 9.2), and if the Time Base update frequency is constant, the period would be

\[ T_{DEC} = \frac{2^{32} \times 32}{1 \text{ GHz}} = 137 \text{ seconds}. \]

The Guest Decrementer counts down.

The operation of the Guest Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Guest Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Guest Decrementer has no effect on the accuracy of the Time Base.
3. Copying the contents of a GPR to the Guest Decrementer replaces the contents of the Guest Decrementer with the contents of the GPR.

9.4.1 Writing and Reading the Guest Decrementer

The contents of the Decrementer can be read or written using the \texttt{mfspr} and \texttt{mtspr} instructions, both of which are supervisor privileged.

Copying the Guest Decrementer to a GPR has no effect on the Guest Decrementer contents or on the interrupt mechanism.

9.4.2 Guest Decrementer Events

A Guest Decrementer event occurs when a decrement occurs on a Guest Decrementer value of 0x0000_0001.

Upon the occurrence of a Guest Decrementer event, the Guest Decrementer may be reloaded from a 32-bit Guest Decrementer Auto-Reload Register (GDECAR). See see Section 9.6. Upon the occurrence of a Guest Decrementer event, the Guest Decrementer has the following basic modes of operation.

---

**Programming Note**

In systems that change the Time Base update frequency for purposes such as power management, the Guest Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Guest Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.
Decrement to one and stop on zero

If GTCRARE=0, GTSRDIS is set to 1, the value 0x0000_0000 is then placed into the GDEC, and the Guest Decrementer stops decrementing.

A Guest Decrementer interrupt occurs when no higher priority interrupt exists, a Guest Decrementer exception exists, and the exception is enabled. The interrupt is enabled by GTCR DIE=1 and (MSREE=1 or MSRGS=1). See Section 7.6.12, “Decrementer Interrupt” on page 1173 for details of register behavior caused by the Guest Decrementer interrupt.

Decrement to one and auto-reload

If GTCRARE=1, GTSRDIS is set to 1, the contents of the Guest Decrementer Auto-Reload Register is then placed into the GDEC, and the Guest Decrementer continues decrementing from the reloaded value.

A Guest Decrementer interrupt occurs when no higher priority interrupt exists, a Guest Decrementer exception exists, and the exception is enabled. The interrupt is enabled by GTCR DIE=1 and (MSREE=1 or MSRGS=1). See Section 7.6.12, “Decrementer Interrupt” on page 1173 for details of register behavior caused by the Guest Decrementer interrupt.

Forcing the Guest Decrementer to 0 using the mtsp r instruction will not cause a Guest Decrementer exception; however, decrementing which was in progress at the instant of the mtsp r may cause the exception. To eliminate the Guest Decrementer as a source of exceptions, set GTCRDIE to 0 (clear the Guest Decrementer Interrupt Enable bit).

If it is desired to eliminate all Guest Decrementer activity, the procedure is as follows:
1. Write 0 to GTCR DIE. This will prevent Guest Decrementer activity from causing exceptions.
2. Write 0 to GTCRARE to disable the Guest Decrementer auto-reload.
3. Write 0 to Guest Decrementer. This will halt Guest Decrementer decrementing. While this action will not cause a Guest Decrementer exception to be set in GTSRDIS, a near simultaneous decrement may have done so.
4. Write 1 to GTSRDIS. This action will clear GTSRDIS to 0 (see Section 9.8.1 on page 1207). This will clear any Guest Decrementer exception which may be pending. Because the Guest Decrementer is frozen at zero, no further Guest Decrementer events are possible.

If the auto-reload feature is disabled (GTCRARE=0), then once the Guest Decrementer decrements to zero, it will stay there until software reloads it using the mtsp r instruction.

On reset, GTCRARE is set to 0. This disables the auto-reload feature.

Programming Note

mtsp r RT,DEC should be used to read GDEC in guest supervisor state. mtsp r DEC,RS should be used to write GDEC in guest supervisor state.

9.5 Decrementer Auto-Reload Register

The Decrementer Auto-Reload Register is a 32-bit register as shown below.

| DECAR | 32 | 63 |

Figure 77. Decrementer Auto-Reload Register

Bits of the Decrementer Auto-Reload register are numbered 32 (most-significant bit) to 63 (least-significant bit). The Decrementer Auto-Reload Register is provided to support the auto-reload feature of the Decrementer. See Section 9.3.2

The contents of the Decrementer Auto-Reload Register cannot be read. The contents of bits 32:63 of register RS can be written to the Decrementer Auto-Reload Register using the mtsp r instruction.

This register is hypervisor privileged.

9.6 Guest Decrementer Auto-Reload Register [Category:Embedded.Hypervisor]

The Guest Decrementer Auto-Reload Register is a 32-bit register as shown below.

| GDECAR | 32 | 63 |

Figure 78. Guest Decrementer Auto-Reload Register

Bits of the Guest Decrementer Auto-Reload Register are numbered 32 (most-significant bit) to 63 (least-significant bit). The Guest Decrementer Auto-Reload Register is provided to support the auto-reload feature of the Guest Decrementer. See Section 9.4.2.

The contents of the Guest Decrementer Auto-Reload Register cannot be read. The contents of bits 32:63 of register RS can be written to the Guest Decrementer Auto-Reload Register using the mtsp r instruction.

This register is hypervisor privileged.
**Programming Note**

`mtspr DECAR,RS` should be used to write GDE-CAR in guest supervisor state. Hypervisor software should emulate the accesses for the guest.
9.7 Timer Control Register

The Timer Control Register (TCR) is a 32-bit register. Timer Control Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Control Register controls Decrementer (see Section 9.3), Fixed-Interval Timer (see Section 9.9), and Watchdog Timer (see Section 9.11) options.

The relationship of the Timer facilities to the TCR and TB is shown in the figure below.

This register is hypervisor privileged. In guest supervisor state, the access to the TCR is mapped to the GTCR.

The contents of the TCR can be read using the *mfspr* instruction. The contents of bits 32:63 of register RS can be written to the Timer Control Register using the *mtspr* instruction.

The contents of the TCR are defined below:

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:33</td>
<td><strong>Watchdog Timer Period</strong> (WP) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td></td>
<td>Specifies one of 4 bit locations of the Time Base used to signal a Watchdog Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Watchdog Timer period are implementation-dependent.</td>
</tr>
<tr>
<td>34:35</td>
<td><strong>Watchdog Timer Reset Control</strong> (WRC) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td></td>
<td>00 No Watchdog Timer reset will occur.</td>
</tr>
<tr>
<td></td>
<td>01-11 Force thread to be reset on second time-out of Watchdog Timer. The exact function of any of these settings is implementation-dependent.</td>
</tr>
<tr>
<td></td>
<td>TCRWRC resets to 0b00.</td>
</tr>
</tbody>
</table>

**Architecture Note**

In previous versions of the architecture, it was not possible for software to clear WRC. That limitation has been removed.

**Watchdog Timer Interrupt Enable** (WIE) (see Section 9.11 on page 1208)

0 Disable Watchdog Timer interrupt
1 Enable Watchdog Timer interrupt

**Decrementer Interrupt Enable** (DIE) (see Section 9.3 on page 1199)

0 Disable Decrementer interrupt
1 Enable Decrementer interrupt

38:39 Fixed-Interval Timer Period (FP) (see Section 9.9 on page 1208)

Specifies one of 4 bit locations of the Time Base used to signal a Fixed-Interval Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Fixed-Interval Timer period are implementation-dependent.

40 Fixed-Interval Timer Interrupt Enable (FIE) (see Section 9.9 on page 1208)

0 Disable Fixed-Interval Timer interrupt
1 Enable Fixed-Interval Timer interrupt

41 Auto-Reload Enable (ARE)

0 Disable auto-reload of the Decrementer
1 Enable auto-reload of the Decrementer

Decrementer exception is presented (i.e. TSRDIS is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The next value placed in the Decrementer is the value 0x0000_0000.

If Category: Embedded.Hypervisor is supported, when (MSR EE=1 or MSR GS=1), TCRDIE=1, and TSRDIS=1, a Decrementer interrupt is taken.

If Category: Embedded.Hypervisor is not supported, when MSR EE=1, TCRDIE=1, and TSRDIS=1, a Decrementer interrupt is taken. Software must reset TSRDIS.

1 Enable auto-reload of the Decrementer

Decrementer exception is presented (i.e. TSRDIS is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The contents of the Decrementer Auto-Reload Register is placed in the Decrementer. The Decrementer resumes decrementing.

If Category: Embedded.Hypervisor is supported, when (MSR EE=1 or MSR GS=1), TCRDIE=1, and TSRDIS=1, a Decrementer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR EE=1, TCRDIE=1, and TSRDIS=1, a Decrementer interrupt is taken. Software must reset TSRDIS.

42 Implementation-dependent

43:63 Reserved

9.7.1 Timer Status Register

The Timer Status Register (TSR) is a 32-bit register. Timer Status Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Status Register contains status on timer events and the most recent Watchdog Timer-initiated thread reset.

The Timer Status Register is set via hardware, and read and cleared via software. The contents of the Timer Status Register can be read using the msfpr instruction. Bits in the Timer Status Register can be cleared using the msfpr instruction. Clearing is done by writing bits 32:63 of a General Purpose Register to the Timer Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The contents of the TSR are defined below:

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Enable Next Watchdog Timer (ENW) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td>0</td>
<td>Action on next Watchdog Timer time-out is to set TSRN</td>
</tr>
<tr>
<td>1</td>
<td>Action on next Watchdog Timer time-out is governed by TSRW</td>
</tr>
<tr>
<td>33</td>
<td>Watchdog Timer Interrupt Status (WIS) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td>0</td>
<td>A Watchdog Timer event has not occurred.</td>
</tr>
<tr>
<td>1</td>
<td>A Watchdog Timer event has occurred. If Category: Embedded.Hypervisor is supported, when (MSR CE=1 or MSR GS=1) and TCRWIE=1, a Watchdog Timer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSRCE=1 and TCRWIE=1, a Watchdog Timer interrupt is taken.</td>
</tr>
<tr>
<td>34:35</td>
<td>Watchdog Timer Reset Status (WRS) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td>00</td>
<td>No Watchdog Timer reset has occurred.</td>
</tr>
<tr>
<td>01</td>
<td>Implementation-dependent reset information.</td>
</tr>
<tr>
<td>10</td>
<td>Implementation-dependent reset information.</td>
</tr>
<tr>
<td>11</td>
<td>Implementation-dependent reset information.</td>
</tr>
<tr>
<td>36</td>
<td>Decrementer Interrupt Status (DIS) (see Section 9.3.2 on page 1199)</td>
</tr>
<tr>
<td>0</td>
<td>A Decrementer event has not occurred.</td>
</tr>
<tr>
<td>1</td>
<td>A Decrementer event has occurred. If Category: Embedded.Hypervisor is supported, when (MSR EE=1 or MSR GS=1) and TCRDIE=1, a Decrementer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR EE=1 and TCRDIE=1, a Decrementer interrupt is taken.</td>
</tr>
</tbody>
</table>
37 **Fixed-Interval Timer Interrupt Status** (FIS)  
(see Section 9.9 on page 1208)

0 A Fixed-Interval Timer event has not occurred.
1 A Fixed-Interval Timer event has occurred. If Category: Embedded.Hypervisor is supported, when (MSR EE=1 or MSR GS=1) and TCR FIE=1, a Fixed-Interval Timer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR EE=1 and TCR FIE=1, a Fixed-Interval Timer interrupt is taken.

38:63 Reserved
This register is hypervisor privileged. In guest supervisor state, the access to the TSR is mapped to the GTSR.
9.8 Guest Timer Control Register [Category: Embedded.Hypervisor]

The Guest Timer Control Register (GTCR) is a 32-bit register. Guest Timer Control Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Guest Timer Control Register controls Guest Decrementer (see Section 9.4), Guest Fixed-Interval Timer (see Section 9.10), and Watchdog Timer (see Section 9.11) options.

The relationship of the Guest Timer facilities to the GTCR and TB is shown in the figure below.

This register is supervisor privileged.

The contents of the Guest Timer Control Register can be read using the \texttt{mfspr} instruction. The contents of bits 32:63 of register RS can be written to the Timer Control Register using the \texttt{mtspr} instruction.

The contents of the GTCR are defined below:

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:33</td>
<td>\textit{Guest Watchdog Timer Period} (WP) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td></td>
<td>Specifies one of 4 bit locations of the Time Base used to signal a Guest Watchdog Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Guest Watchdog Timer period are implementation-dependent.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34:35</td>
<td>\textit{Guest Watchdog Timer Reset Control} (WRC) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No Guest Watchdog Timer reset will occur</td>
</tr>
<tr>
<td></td>
<td>\texttt{GTCRWRC} resets to 0b00.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-11</td>
<td>Force thread to signal a Watchdog Timer exception to the hypervisor on second time</td>
</tr>
</tbody>
</table>

\textbf{Architecture Note:} In previous versions of the architecture, it was not possible for software to clear WRC. That limitation has been removed.

\textbf{Guest Watchdog Timer Interrupt Enable} (WIE) (see Section 9.11 on page 1208)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>\textit{Guest Watchdog Timer Interrupt Enable} (WIE) (see Section 9.11 on page 1208)</td>
</tr>
<tr>
<td>0</td>
<td>Disable Guest Watchdog Timer interrupt</td>
</tr>
<tr>
<td>1</td>
<td>Enable Guest Watchdog Timer interrupt</td>
</tr>
</tbody>
</table>

\textbf{Guest Decrementer Interrupt Enable} (DIE) (see Section 9.3 on page 1199)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>\textit{Guest Decrementer Interrupt Enable} (DIE) (see Section 9.3 on page 1199)</td>
</tr>
<tr>
<td>0</td>
<td>Disable Guest Decrementer interrupt</td>
</tr>
</tbody>
</table>
1 Enable Guest Decrementer interrupt

38:39 Guest Fixed-Interval Timer Period (FP) (see Section 9.9 on page 1208)

Specifies one of 4 bit locations of the Time Base used to signal a Guest Fixed-Interval Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Guest Fixed-Interval Timer period are implementation-dependent.

40 Guest Fixed-Interval Timer Interrupt Enable (FIE) (see Section 9.9 on page 1208)

0 Disable Guest Fixed-Interval Timer interrupt
1 Enable Guest Fixed-Interval Timer interrupt

41 Guest Auto-Reload Enable (ARE)

0 Disable auto-reload of the Guest Decrementer

Guest Decrementer exception is presented (i.e. GTSRDIS is set to 1) when the Guest Decrementer is decremented from a value of 0x0000_0001. The next value placed in the Guest Decrementer is the value 0x0000_0000. When (MSRE=1 and MSRGS=1), GTCRDIE=1, and GTSRDIS=1, a Guest Decrementer interrupt is taken. Software must reset GTSRDIS.

1 Enable auto-reload of the Guest Decrementer

Guest Decrementer exception is presented (i.e. GTSRDIS is set to 1) when the Guest Decrementer is decremented from a value of 0x0000_0001. The contents of the Guest Decrementer Auto-Reload Register is placed in the Guest Decrementer. When (MSRE=1 and MSRGS=1), GTCRDIE=1, and GTSRDIS=1, a Guest Decrementer interrupt is taken. Software must reset GTSRDIS.

42 Implementation-dependent

43:63 Reserved

Programming Note

*mspr* RT,TCR should be used to read GTCR in guest supervisor state. *mtspr* TCR,RS should be used to write GTCR in guest supervisor state.

9.8.1 Guest Timer Status Register

[Category: Embedded.Hypervisor]

The Guest Timer Status Register (GTSR) is a 32-bit register. Guest Timer Status Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Guest Timer Status Register contains status on timer events and the most recent Watchdog Timer-initiated thread reset.

The Guest Timer Status Register is set via hardware, and read and cleared via software. The contents of the Guest Timer Status Register can be read using the *mfspr* instruction. Bits in the Guest Timer Status Register can be cleared using the *mtspr* instruction. Clearing is done by writing bits 32:63 of a General Purpose Register to the Guest Timer Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Guest Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The contents of the GTSR are defined below:

**Bit(s)** | **Description**
--- | ---
32 | Enable Next Guest Watchdog Timer (ENW) (see Section 9.11 on page 1208)
0 | Action on next Guest Watchdog Timer time-out is to set GTSRENW
1 | Action on next Guest Watchdog Timer time-out is governed by GTSRWIS
33 | Guest Watchdog Timer Interrupt Status (WIS) (see Section 9.11 on page 1208)
0 | A Guest Watchdog Timer event has not occurred.
1 | A Guest Watchdog Timer event has occurred. When (MSR CE=1 and MSRGS=1) and GTCRWIE=1, a Guest Watchdog Timer interrupt is taken.
34:35 | Guest Watchdog Timer Reset Status (WRS) (see Section 9.11 on page 1208)
00 | No Guest Watchdog Timer reset has occurred.
01 | Implementation-dependent reset information.
10 | Implementation-dependent reset information.
11 | Implementation-dependent reset information.
36 | Guest Decrementer Interrupt Status (DIS) (see Section 9.4.2 on page 1200)
0 | A Guest Decrementer event has not occurred.
1 | A Guest Decrementer event has occurred. When MSR EE=1 and MSRGS=1 and GTCRDIE=1, a Guest Decrementer interrupt is taken.
37 | Guest Fixed-Interval Timer Interrupt Status (FIS) (see Section 9.10 on page 1208)
0 | A Guest Fixed-Interval Timer event has not occurred.
A Guest Fixed-Interval Timer event has occurred. When (MSR_EE=1 and MSR_GS=1) and GTCR_FIE=1, a Guest Fixed-Interval Timer interrupt is taken.

This register is supervisor privileged.

Programming Note

\texttt{mtspr RT,TSR} should be used to read GTSR in guest supervisor state. \texttt{mtspr TSR,RS} should be used to write GTSR in guest supervisor state.

### 9.8.2 Guest Timer Status Register Write Register (GTSRWR) [Category: Embedded.Hypervisor]

The Guest Timer Status Register Write Register (GTSRWR) allows a hypervisor state program to write the contents of the Guest Timer Status Register (see Section 9.8.1). The format of the GTSRWR is shown in Figure 81 below.

![Figure 81. Guest Timer Status Register Write Register](image)

The GTSRWR is provided as a means to restore the contents of the GTSR on a partition switch.

Writing GTSRWR changes the value in the GTSR. Writing non-zero bits may cause a Guest Decrementer or Fixed-Interval Timer exception.

This register is hypervisor privileged.

Programming Note

Hypervisors must ensure that a partition swap does not cause missing timer events to occur in guests. Upon partition restore, the hypervisor must set the appropriate status conditions in the GTSR.

### 9.9 Fixed-Interval Timer

The Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload Decrementer, except that there are fewer selections of interrupt period available. The Fixed-Interval Timer exception occurs on 0 to 1 transitions of a selected bit from the Time Base (see Section 9.7).

The Fixed-Interval Timer exception is logged by GTSRFIS. A Fixed-Interval Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Fixed-Interval Timer exception exists (TSRFIE=1), and the exception is enabled. If category mbedded.Hypervisor is supported, the interrupt is enabled by TCRFIE=1 and (MSRE= 1 or MSRGS=1). Otherwise, the interrupt is enabled by TCRFIE=1 and MSREE=1. See Section 7.6.14 on page 1174 for details of register behavior caused by the Fixed-Interval Timer interrupt.

Note that a Fixed-Interval Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an \texttt{mtspr} instruction that writes a 1 to the bit when its previous value was 0.

### 9.10 Guest Fixed-Interval Timer

[Category: Embedded.Hypervisor]

The Guest Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload Guest Decrementer, except that there are fewer selections of interrupt period available. The Guest Fixed-Interval Timer exception occurs on 0 to 1 transitions of a selected bit from the Time Base (see Section 9.7).

The Guest Fixed-Interval Timer exception is logged by GTSRFIS. A Guest Fixed-Interval Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Fixed-Interval Timer exception exists (GTSRFIS=1), and the exception is enabled. The interrupt is enabled by GTCR_FIE=1 and (MSRE = 1 and MSRGS=1). See Section 7.6.14 on page 1174 for details of register behavior caused by the Fixed-Interval Timer interrupt.

Note that a Guest Fixed-Interval Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an \texttt{mtspr} instruction that writes a 1 to the bit when its previous value was 0.

### 9.11 Watchdog Timer

The Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the Time Base (Section 9.7).

When a Watchdog Timer time-out occurs while Watchdog Timer Interrupt Status is clear (TSRWIS = 0) and the next Watchdog Time-out is enabled (TSRENW = 1), a Watchdog Timer exception is generated and logged by setting TSRWIS to 1. This is referred to as a Watchdog Timer First Time Out. A Watchdog Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Watchdog Timer excep-
tion exists \( \text{TSRWIS}=1 \), and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by \( \text{TCR}_{\text{WIE}}=1 \) and \( \text{MSR}_{\text{CE}} = 1 \) or \( \text{MSR}_{\text{GS}}=1 \). Otherwise, the interrupt is enabled by \( \text{TCR}_{\text{WIE}}=1 \) and \( \text{MSR}_{\text{CE}}=1 \). See Section 7.6.16 on page 1175 for details of register behavior caused by the Watchdog Timer Interrupt. The purpose of the Watchdog Timer First time-out is to give an indication that there may be a problem and give the system a chance to perform corrective action or capture a failure before a reset occurs from the Watchdog Timer Second time-out as explained further below.

Note that a Watchdog Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an \textit{mtspr} instruction that writes a 1 to the bit when its previous value was 0.

When a Watchdog Timer time-out occurs while \( \text{TSRWIS} = 1 \) and \( \text{TSRENW} = 1 \), a thread reset occurs if it is enabled by a non-zero value of the Watchdog Reset Control field in the Timer Control Register (TCR\(_{\text{WRC}}\)). This is referred to as a Watchdog Timer Second Time Out. The assumption is that \( \text{TSRWIS} \) was not cleared because the thread was unable to execute the Watchdog Timer interrupt handler, leaving reset as the only available means to restart the system.

A more complete view of Watchdog Timer behavior is afforded by Figure 82 and Figure 83, which describe the Watchdog Timer state machine and Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.

---

**Figure 82. Watchdog State Machine**

When a Watchdog Timer time-out occurs while \( \text{TSRWIS} = 1 \) and \( \text{TSRENW} = 1 \), a thread reset occurs if it is enabled by a non-zero value of the Watchdog Reset Control field in the Timer Control Register (TCR\(_{\text{WRC}}\)). This is referred to as a Watchdog Timer Second Time Out. The assumption is that \( \text{TSRWIS} \) was not cleared because the thread was unable to execute the Watchdog Timer interrupt handler, leaving reset as the only available means to restart the system.

A more complete view of Watchdog Timer behavior is afforded by Figure 82 and Figure 83, which describe the Watchdog Timer state machine and Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.
Figure 83. Watchdog Timer Controls

The controls described in the above table imply three different modes of operation that a programmer might select for the Watchdog Timer. Each of these modes assumes that TCRWRC has been set to allow thread reset by the Watchdog facility:

1. Always take the Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Watchdog Timer interrupt caused by a first time-out is used to clear TSRWIS so a second time-out never occurs. TSRENW is not cleared, thereby allowing the next time-out to cause another interrupt.

2. Always take the Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) is used to repeatedly clear TSRWIS such that a first-time-out exception is avoided, and thus no Watchdog Timer interrupt occurs. Once TSRWIS has been cleared, software has between one and two full Watchdog periods before a Watchdog exception will be posted in TSRWIS. If this occurs before the software is able to clear TSNW again, a Watchdog Timer interrupt will occur. In this case, the Watchdog Timer interrupt handler will then clear both TSNW and TSRWIS, in order to (hopefully) avoid the next Watchdog Timer interrupt.

3. Never take the Watchdog Timer interrupt. In this mode, Watchdog Timer interrupts are disabled (via TCRWIE=0), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) to repeatedly clear TSRWIS such that a second time-out is avoided, and thus no reset occurs. TSRWIS is not cleared, thereby allowing the next time-out to set TSRWIS again. The recurring code loop must have a period which is less than one Watchdog Timer period in order to guarantee that a Watchdog Timer reset will not occur.

9.12 Guest Watchdog Timer

[Category: Embedded.Hypervisor]

The Guest Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Guest Watchdog time-outs occur on 0 to 1 transitions of selected bits from the Time Base (Section 9.7).

When a Guest Watchdog Timer time-out occurs while Guest Watchdog Timer Interrupt Status is clear (GTSRWIS = 0) and the next Guest Watchdog Time-out is enabled (GTSRENW = 1), a Guest Watchdog Timer exception is generated and logged by setting GTSRWIS to 1. This is referred to as a Guest Watchdog Timer First Time Out. A Guest Watchdog Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Watchdog Timer exception exists (GTSRWIS=1), and the exception is enabled. The interrupt is enabled by GTCTRWE=1 and (MSRCE = 1 and MSRGS=1). See Section 7.6.17 on page 1176 for details of register behavior caused by the Guest Watchdog Timer Interrupt. The purpose of the Guest Watchdog Timer First-time-out is to give an indication that there may be problem and give the system a chance to perform corrective action or capture a failure before a virtualized reset (a Watchdog Timer exception in the hypervisor) occurs from the Guest Watchdog Timer Second time-out as explained further below.
Note that a Guest Watchdog Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an `mtspr` instruction that writes a 1 to the bit when its previous value was 0.

A Guest Watchdog Timer Second Time Out results when a Guest Watchdog Timer time-out occurs while `GTSRWIS = 1`, `GTSRENW = 1`, and a non-zero value is present in the Guest Watchdog Reset Control field in the Guest Timer Control Register (GTCRWRC). In this case, a Watchdog Timer Interrupt is directed toward the hypervisor and the value set in GTSRWRS reflects the virtualized reset condition. The assumption is that GTSRWIS was not cleared because the guest was unable to execute the Guest Watchdog Timer interrupt handler, leaving a virtualized reset as the only available means to stop or restart the guest.

A more complete view of Guest Watchdog Timer behavior is afforded by Figure 84 and Figure 85, which describe the Guest Watchdog Timer state machine and Guest Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.

![Figure 84. Guest Watchdog State Machine](image)

<table>
<thead>
<tr>
<th>Enable Next WDT (GTSRENW)</th>
<th>WDT Status (GTSRWIS)</th>
<th>Action when timer interval expires</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Set Enable Next Guest Watchdog Timer (GTSRENW=1).</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Set Enable Next Guest Watchdog Timer (GTSRENW=1).</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Set Guest Watchdog Timer interrupt status bit (GTSRWIS=1). If Guest Watchdog Timer interrupt is enabled (GTCRWIE=1 and (MSRCE=1 and MSRGS=1)), then interrupt.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Cause Guest Watchdog Timer virtualized reset action specified by GTCRWRC. Virtualized reset will copy pre-reset GTCRWRC into GTSRWRS, then clear GTCRWRC.</td>
</tr>
</tbody>
</table>

![Figure 85. Guest Watchdog Timer Controls](image)

The controls described in the above table imply three different modes of operation that a programmer might select for the Guest Watchdog Timer. Each of these modes assumes that GTCRWRC has been set on a thread to allow virtualized reset by the Guest Watchdog facility.
1. Always take the Guest Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Guest Watchdog Timer interrupt caused by a first time-out is used to clear GTSRWIS so a second time-out never occurs. GTSRENW is not cleared, thereby allowing the next time-out to cause another interrupt.

2. Always take the Guest Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Guest Fixed-Interval Timer interrupt handler) is used to repeatedly clear GTSRENW such that a first time-out exception is avoided, and thus no Guest Watchdog Timer interrupt occurs. Once GTSRENW has been cleared, software has between one and two full Guest Watchdog periods before a Guest Watchdog exception will be posted in GTSRWIS. If this occurs before the software is able to clear GTSRENW again, a Guest Watchdog Timer interrupt will occur. In this case, the Guest Watchdog Timer interrupt handler will then clear both GTSRENW and GTSRWIS, in order to (hopefully) avoid the next Guest Watchdog Timer interrupt.

3. Never take the Guest Watchdog Timer interrupt. In this mode, Guest Watchdog Timer interrupts are disabled (via GTCRWIE=0), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Guest Fixed-Interval Timer interrupt handler) to repeatedly clear GTSRWIS such that a second time-out is avoided, and thus no virtualized reset occurs. GTSRENW is not cleared, thereby allowing the next time-out to set GTSRWIS again. The recurring code loop must have a period which is less than one Guest Watchdog Timer period in order to guarantee that a Guest Watchdog Timer virtualized reset will not occur.

9.13 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers upon a debug event. Whenever a debug event is set in the Debug Status Register, all timers will be frozen by preventing the Time Base from incrementing. This allows a debugger to simulate the appearance of ‘real time’, even though the application has been temporarily ‘halted’ to service the debug event. See the description of bit 63 of the Debug Control Register 0 (Freeze Timers on Debug Event or DBCR0FT) in Section 10.5.1.1 on page 1221.
Chapter 10. Debug Facilities

10.1 Overview

Debug facilities are provided to enable hardware and software debug functions, such as instruction and data breakpoints and program single stepping. The debug facilities consist of a set of Debug Control Registers (DBCR0, DBCR1, and DBCR2) (see Section 10.5.1 on page 1221), a set of Address Compare Registers (IAC1, IAC2, IAC3, IAC4, DAC1, and DAC2), (see Section 10.4.3, Section 10.4.4, and Section 10.4.5), a Debug Status Register (DBSR) (see Section 10.5.2) for enabling and recording various kinds of debug events, and a special Debug interrupt type built into the interrupt mechanism (see Section 7.6.20). The debug facilities also provide a mechanism for software-controlled thread reset, and for controlling the operation of the timers in a debug environment.

The mfspr and mtspr instructions (see Section 5.4.1) provide access to the registers of the debug facilities.

In addition to the facilities described here, implementations will typically include debug facilities, modes, and access mechanisms which are implementation-specific. For example, implementations will typically provide access to the debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

10.2 Internal Debug Mode

Debug events include such things as instruction and data breakpoints. These debug events cause status bits to be set in the Debug Status Register. The existence of a set bit in the Debug Status Register is considered a Debug exception. Debug exceptions, if enabled, will cause Debug interrupts.

There are two different mechanisms that control whether Debug interrupts are enabled. The first is the MSRDE bit, and this bit must be set to 1 to enable Debug interrupts. The second mechanism is an enable bit in the Debug Control Register 0 (DBCR0). This bit is the Internal Debug Mode bit (DBCR0IDM), and it must also be set to 1 to enable Debug interrupts.

When DBCR0IDM = 1, the thread is in Internal Debug Mode. In this mode, debug events will (if also enabled by MSRDE) cause Debug interrupts. Software at the Debug interrupt vector location will thus be given control upon the occurrence of a debug event, and can access (via the normal instructions) all architected resources. In this fashion, debug monitor software can control the thread and gather status, and interact with debugging hardware.

When the thread is not in Internal Debug Mode (DBCR0IDM = 0), debug events may still occur and be recorded in the Debug Status Register. These exceptions may be monitored via software by reading the Debug Status Register (using mfspr), or may eventually cause a Debug interrupt if later enabled by setting DBCR0IDM = 1 (and MSRDE = 1). Behavior when debug events occur while DBCR0IDM = 0 is implementation-dependent.

10.3 External Debug Mode [Category: Embedded.Enhanced Debug]

The External Debug Mode is a mode in which external facilities can control execution and access registers and other resources. These facilities are defined as the external debug facilities and are not defined here, however some instructions and registers share internal and external debug roles and are briefly described as necessary.

A dnh instruction is provided to stop instruction fetching and execution and allow the thread to be managed by an external debug facility. After the dnh instruction is executed, instructions are not fetched, interrupts are not taken, and the thread does not execute instructions.

10.4 Debug Events

Debug events are used to cause Debug exceptions to be recorded in the Debug Status Register (see Section 10.5.2). In order for a debug event to be enabled to set a Debug Status Register bit and thereby
cause a Debug exception, the specific event type must be enabled by a corresponding bit or bits in the Debug Control Register DBCR0 (see Section 10.5.1.1), DBCR1 (see Section 10.5.1.2), or DBCR2 (see Section 10.5.1.3), in most cases; the Unconditional Debug Event (UDE) is an exception to this rule. Once a Debug Status Register bit is set, if Debug interrupts are enabled by MSRDE, a Debug interrupt will be generated.

To prevent spurious hypervisor debug events from occurring when a guest has been permitted to use the Debug facilities, if the thread is in hypervisor state (MSRDE=0) and debug events are disabled for hypervisor (EPCRDUVD=1), no debug events are allowed to occur except for the Unconditional Debug Event. It is implementation-dependent whether the Unconditional Debug Event is allowed to occur in hypervisor state when EPCRDUVD=1.

Certain debug events are not allowed to occur when MSRDE=0. In such situations, no Debug exception occurs and thus no Debug Status Register bit is set. Other debug events may cause Debug exceptions and set Debug Status Register bits regardless of the state of MSRDE. The associated Debug interrupts that result from such Debug exceptions will be delayed until MSRDE=1, provided the exceptions have not been cleared from the Debug Status Register in the meantime.

Any time that a Debug Status Register bit is allowed to be set while MSRDE=0, a special Debug Status Register bit, Imprecise Debug Event (DBSR_IDE), will also be set. DBSR_IDE indicates that the associated Debug exception bit in the Debug Status Register was set while Debug interrupts were disabled via the MMSRDE bit. Debug interrupt handler software can use this bit to determine whether the address recorded in CSRR0/DSRR0 should be interpreted as the address associated with the instruction causing the Debug exception, or simply the address of the instruction after the one which set the MSRDE bit, thereby enabling the delayed Debug interrupt.

Debug interrupts are ordered with respect to other interrupt types (see Section 7.8 on page 179). Debug exceptions are prioritized with respect to other exceptions (see Section 7.9 on page 183).

There are eight types of debug events defined:

1. Instruction Address Compare debug events
2. Data Address Compare debug events
3. Trap debug events
4. Branch Taken debug events
5. Instruction Complete debug events
6. Interrupt Taken debug events
7. Return debug events
8. Unconditional debug events
9. Critical Interrupt Taken debug events [Category: Embedded.Enhanced Debug]
10. Critical Interrupt Return debug events [Category: Embedded.Enhanced Debug]
There are two classes of debug exception types:

Type 1: exception before instruction

Type 2: exception after instruction

Almost all debug exceptions fall into the first type. That is, they all take the interrupt upon encountering an instruction having the exception without updating any architectural state (other than DBSR, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug], CSRR1/DSRR1 [Category: Embedded.Enhanced Debug], MSR) for that instruction.

The CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] for this type of exception points to the instruction that encountered the exception. This includes IAC, DAC, branch taken, etc.

The only exception which fall into the second type is the instruction complete debug exception. This exception is taken upon completing and updating one instruction and then pointing CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] to the next instruction to execute.

To make forward progress for any Type 1 debug exception one does the following:

1. Software sets up Type 1 exceptions (e.g. branch taken debug exceptions) and then returns to normal program operation
2. Hardware takes Debug interrupt upon the first branch taken Debug exception, pointing to the branch with CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] to the next instruction to execute.
3. Software, in the debug handler, sees the branch taken exception type, does whatever logging/analysis it wants to, then clears all debug event enables in the DBCR except for the instruction complete debug event enable.
4. Software does an rfci or rfdi [Category: Embedded.Enhanced Debug].
5. Hardware would execute and complete one instruction (the branch taken in this case), and then take a Debug interrupt with CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] pointing to the target of the branch.
6. Software would see the instruction complete interrupt type. It clears the instruction complete event enable, then enables the branch taken interrupt event again.
7. Software does an rfci or rfdi [Category: Embedded.Enhanced Debug].
8. Hardware resumes on the target of the taken branch and continues until another taken branch, in which case we end up at step 2 again.

This, at first, seems like a double tax (i.e. 2 debug interrupts for every instance of a Type 1 exception), but there doesn't seem like any other clean way to make forward progress on Type 1 debug exceptions. The only other way to avoid the double tax is to have the debug handler routine actually emulate the instruction pointed to for the Type 1 exceptions, determine the next instruction that would have been executed by the interrupted program flow and load the CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] with that address and do an rfci/rfdi [Category: Embedded.Enhanced Debug]; this is probably not faster.

10.4.1 Instruction Address Compare Debug Event

One or more Instruction Address Compare debug events (IAC1, IAC2, IAC3 or IAC4) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR0, DBCR1, IAC1, IAC2, IAC3, and IAC4 Registers.

Instruction Address Compare User/Supervisor Mode

DBCR1IAC1US specifies whether IAC1 debug events can occur in user mode or supervisor mode, or both.

DBCR1IAC2US specifies whether IAC2 debug events can occur in user mode or supervisor mode, or both.

DBCR1IAC3US specifies whether IAC3 debug events can occur in user mode or supervisor mode, or both.

DBCR1IAC4US specifies whether IAC4 debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

DBCR1IAC1ER specifies whether effective addresses, real addresses, effective addresses and MSR1S=0, or effective addresses and MSR1S=1 are used in determining an address match on IAC1 debug events.

DBCR1IAC2ER specifies whether effective addresses, real addresses, effective addresses and MSR1S=0, or
effective addresses and MSR$_{IS}=1$ are used in determining an address match on IAC2 debug events.

DBCR$_{IAC3ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR$_{IS}=0$, or effective addresses and MSR$_{IS}=1$ are used in determining an address match on IAC3 debug events.

DBCR$_{IAC4ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR$_{IS}=0$, or effective addresses and MSR$_{IS}=1$ are used in determining an address match on IAC4 debug events.

### Instruction Address Compare Mode

DBCR$_{IAC12M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC1 or IAC2, whether the address must be inside a specific range specified by the IAC1 and IAC2 or outside a specific range specified by the IAC1 and IAC2 for an IAC1 or IAC2 debug event to occur.

DBCR$_{IAC34M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC3 Register or IAC4 Register, whether the address must be inside a specific range specified by the IAC3 Register and IAC4 Register or outside a specific range specified by the IAC3 Register and IAC4 Register for an IAC3 or IAC4 debug event to occur.

There are four instruction address compare modes.

#### Exact address compare mode

If the address of the instruction fetch is equal to the value in the enabled IAC Register, an instruction address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

#### Address bit match mode

For IAC1 and IAC2 debug events, if the address of the instruction fetch access, ANDed with the contents of the IAC2, are equal to the contents of the IAC1, also ANDed with the contents of the IAC2, an instruction address match occurs.

For IAC3 and IAC4 debug events, if the address of the instruction fetch, ANDed with the contents of the IAC4, are equal to the contents of the IAC3, also ANDed with the contents of the IAC4, an instruction address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

#### Inclusive address range compare mode

For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the IAC1 and less than the contents of the IAC2, an instruction address match occurs.

For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the IAC3 and less than the contents of the IAC4, an instruction address match occurs.

- For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

### Exclusive address range compare mode

For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is less than the contents of the IAC1 or greater than or equal to the contents of the IAC2, an instruction address match occurs.

For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is less than the contents of the IAC3 or greater than or equal to the contents of the IAC4, an instruction address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

See the detailed description of DBCR0 (see Section 10.5.1.1, “Debug Control Register 0 (DBCR0)” on page 1221) and DBCR1 (see Section 10.5.1.2, “Debug Control Register 1 (DBCR1)” on page 1222) and the modes for detecting IAC1, IAC2, IAC3 and IAC4 debug events. Instruction Address Compare debug events can occur regardless of the setting of MSR$_{DE}$ or DBCR0$_{IDM}$.

When an Instruction Address Compare debug event occurs, the corresponding DBSR$_{IAC1}$, DBSR$_{IAC2}$, DBSR$_{IAC3}$, or DBSR$_{IAC4}$ bit or bits are set to record the debug exception. If MSR$_{DE}=0$, DBSR$_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR$_{DE}=1$ (i.e. Debug interrupts are enabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded. Enhanced Debug] will be set to the address of the excepting instruction.

If MSR$_{DE}=0$ (i.e. Debug interrupts are disabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt).
Later, if the debug exception has not been reset by clearing DBSR\textsubscript{IAC1}, DBSR\textsubscript{IAC2}, DBSR\textsubscript{IAC3}, and DBSR\textsubscript{IAC4}, and MSR\textsubscript{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR\textsubscript{0}/DSRR\textsubscript{0} will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR\textsubscript{DE} to 1. Software in the Debug interrupt handler can observe DBSR IDE to determine how to interpret the value in CSRR\textsubscript{0}/DSRR\textsubscript{0}.

### 10.4.2 Data Address Compare Debug Event

One or more Data Address Compare debug events (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data storage access instruction, and the type, address, and possibly even the data value of the data storage access meet the criteria specified in the Debug Control Register 0, Debug Control Register 2, and the DAC1 and DAC2 Registers.

#### Data Address Compare Read/Write Enable

DBCR\textsubscript{0DAC1} specifies whether DAC1R debug events can occur on read-type data storage accesses and whether DAC1W debug events can occur on write-type data storage accesses.

DBCR\textsubscript{0DAC2} specifies whether DAC2R debug events can occur on read-type data storage accesses and whether DAC2W debug events can occur on write-type data storage accesses.

Indexed-string instructions (\textit{lswx}, \textit{stswx}) for which the XER field specifies zero bytes as the length of the string are treated as no-ops, and are not allowed to cause Data Address Compare debug events.

All Load instructions are considered reads with respect to debug events, while all Store instructions are considered writes with respect to debug events. In addition, the Cache Management instructions, and certain special cases, are handled as follows.

- \textit{dcbt}, \textit{dcbtls}, \textit{dcbilq}, \textit{dcbtep}, \textit{icbt}, \textit{icbtls}, \textit{icbi}, \textit{icblic}, \textit{icbliq}, \textit{dcblc}, and \textit{icbipe} are all considered reads with respect to debug events. Note that \textit{dcbt}, \textit{dcbtep}, and \textit{icbt} are treated as no-operations when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause Debug interrupts, even when they would otherwise have been no-op’ed due to a Data Storage or Data TLB Miss exception.

- \textit{dcbtst}, \textit{dcbtstls}, \textit{dcbtstep}, \textit{dcbz}, \textit{dcbzep}, \textit{dcbi}, \textit{dcbf}, \textit{dcbfep}, \textit{dcba}, \textit{dcbst}, and \textit{dcbstep} are all considered writes with respect to debug events. Note that \textit{dcbf}, \textit{dcbfep}, \textit{dcbst}, and \textit{dcbstep} are considered reads with respect to Data Storage exceptions, since they do not actually change the data at a given address. However, since the execution of these instructions may result in write activity on the data bus, they are treated as writes with respect to debug events. Note also that \textit{dcbtst} and \textit{dcbtstep} are treated as no-operations when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause Debug interrupts, even when they would otherwise have been no-op’ed due to a Data Storage or Data TLB Miss exception.

#### Data Address Compare User/Supervisor Mode

DBCR\textsubscript{2DAC1US} specifies whether DAC1R and DAC1W debug events can occur in user mode or supervisor mode, or both.

DBCR\textsubscript{2DAC2US} specifies whether DAC2R and DAC2W debug events can occur in user mode or supervisor mode, or both.

#### Effective/Real Address Mode

DBCR\textsubscript{2DAC1ER} specifies whether effective addresses, real addresses, effective addresses and MSR\textsubscript{DS}=0, or effective addresses and MSR\textsubscript{DS}=1 are used to in determining an address match on DAC1R and DAC1W debug events.

DBCR\textsubscript{2DAC2ER} specifies whether effective addresses, real addresses, effective addresses and MSR\textsubscript{DS}=0, or effective addresses and MSR\textsubscript{DS}=1 are used to in determining an address match on DAC2R and DAC2W debug events.

#### Data Address Compare Mode

DBCR\textsubscript{2DAC12M} specifies whether all or some of the bits of the address of the data storage access must match the contents of the DAC1 or DAC2, whether the address must be inside a specific range specified by the DAC1 and DAC2 or outside a specific range specified by the DAC1 and DAC2 for a DAC1R, DAC1W, DAC2R or DAC2W debug event to occur.

There are four data address compare modes.

- **Exact address compare mode**
  
  If the 64-bit address of the data storage access is equal to the value in the enabled Data Address Compare Register, a data address match occurs.

Chapter 10. Debug Facilities 1217
For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

- **Address bit match mode**
  If the address of the data storage access, ANDed with the contents of the DAC2, are equal to the contents of the DAC1, also ANDed with the contents of the DAC2, a data address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

- **Inclusive address range compare mode**
  If the 64-bit address of the data storage access is greater than or equal to the contents of the DAC1 and less than the contents of the DAC2, a data address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

- **Exclusive address range compare mode**
  If the 64-bit address of the data storage access is less than the contents of the DAC1 or greater than or equal to the contents of the DAC2, a data address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

The description of DBCR0 (see Section 10.5.1.1) and DBCR2 (see Section 10.5.1.3) and the modes for detecting Data Address Compare debug events. Data Address Compare debug events can occur regardless of the setting of MSRDE or DBCR0IDM.

When an Data Address Compare debug event occurs, the corresponding DBSRDAC1R, DBSRDAC1W, DBSRDAC2R, or DBSRDAC2W bit or bits are set to 1 to record the debug exception. If MSRDE=0, DBSRIDE is also set to 1 to record the imprecise debug event.

If MSRDE=1 (i.e. Debug interrupts are enabled) at the time of the Data Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), the execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSRDE to 1. Software in the Debug interrupt handler can observe DBSRIDE to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

Later, if the debug exception has not been reset by clearing DBSRDAC1R, DBSRDAC1W, DBSRDAC2R, DBSRDAC2W, and MSRDE is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the exceptioning instruction.

If MSRDE=1 (i.e. Debug interrupts are enabled) at the time of the Trap debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the exceptioning instruction.

If MSRDE=0 (i.e. Debug interrupts are disabled) at the time of the Trap debug exception, a Debug interrupt will not occur, and a Trap exception type Program interrupt will occur instead if the trap condition is met. Later, if the debug exception has not been reset by clearing DBSRTR, and MSRDE is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting both MSRDE and DBCR0IDM to 1. Software in the Debug interrupt handler can observe DBSRIDE to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

### 10.4.3 Trap Debug Event

A Trap debug event (TRAP) occurs if DBCR0TRAP=1 (i.e. Trap debug events are enabled) and a Trap instruction (tw, twi, td, tdi) is executed and the conditions specified by the instruction for the trap are met. The event can occur regardless of the setting of MSRDE or DBCR0IDM.

When a Trap debug event occurs, DBSRTR is set to 1 to record the debug exception. If MSRDE=0, DBSRIDE is also set to 1 to record the imprecise debug event.

If MSRDE=1 (i.e. Debug interrupts are enabled) at the time of the Trap debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the exceptioning instruction.

If MSRDE=0 (i.e. Debug interrupts are disabled) at the time of the Trap debug exception, a Debug interrupt will not occur, and a Trap exception type Program interrupt will occur instead if the trap condition is met.

### 10.4.4 Branch Taken Debug Event

A Branch Taken debug event (BRT) occurs if DBCR0BRT=1 (i.e. Branch Taken Debug events are enabled), execution is attempted of a branch instruction whose direction will be taken (that is, either an unconditional branch, or a conditional branch whose branch condition is met), and MSRDE=1.
Branch Taken debug events are not recognized if MSR$_{DE}$=0 at the time of the execution of the branch instruction and thus DBSR$_{IDE}$ can not be set by a Branch Taken debug event. This is because branch instructions occur very frequently. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR$_{DE}$ would result in an inordinate number of imprecise Debug interrupts.

When a Branch Taken debug event occurs, the DBSR$_{BRT}$ bit is set to 1 to record the debug exception and a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

### 10.4.5 Instruction Complete Debug Event

An Instruction Complete debug event (ICMP) occurs if DBCR$_0$ICMP=1 (i.e. Instruction Complete debug events are enabled), execution of any instruction is completed, and MSR$_{DE}$=1. Note that if execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an Instruction Complete debug event. The sc instruction does not fall into the type of an instruction whose execution is suppressed, since the instruction actually completes execution and then generates a System Call interrupt. In this case, the Instruction Complete debug exception will also be set.

Instruction Complete debug events are not recognized if MSR$_{DE}$=0 at the time of the execution of the instruction, DBSR$_{IDE}$ can not be set by an ICMP debug event. This is because allowing the common event of Instruction Completion to be recorded as an exception in the DBSR while Debug interrupts are disabled via MSR$_{DE}$ would mean that the Debug interrupt handler software would receive an inordinate number of imprecise Debug interrupts every time Debug interrupts were re-enabled via MSR$_{DE}$.

When an Instruction Complete debug event occurs, DBSR$_{ICMP}$ is set to 1 to record the debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the instruction after the one causing the Instruction Complete debug exception.

### 10.4.6 Interrupt Taken Debug Event

#### 10.4.6.1 Causes of Interrupt Taken Debug Events

Only base class interrupts and guest class interrupts (Category: Embedded.Hypervisor) can cause an Interrupt Taken debug event. If the Embedded.Enhanced Debug category is not supported or is supported and not enabled, all other interrupts automatically clear MSR$_{DE}$, and thus would always prevent the associated Debug interrupt from occurring precisely. If the Embedded.Enhanced Debug category is supported and enabled, then critical class interrupts do not automatically clear MSR$_{DE}$, but they cause Critical Interrupt Taken debug events instead of Interrupt Taken debug events.

Also, if the Embedded.Enhanced Debug category is not supported or is supported and not enabled, Debug interrupts themselves are critical class interrupts, and thus any Debug interrupt (for any other debug event) would always end up setting the additional exception of DBSR$_{IRPT}$ upon entry to the Debug interrupt handler. At this point, the Debug interrupt handler would be unable to determine whether or not the Interrupt Taken debug event was related to the original debug event.

#### 10.4.6.2 Interrupt Taken Debug Event Description

An Interrupt Taken debug event (IRPT) occurs if DBCR$_0$IRPT=1 (i.e. Interrupt Taken debug events are enabled) and a base class interrupt occurs. Interrupt Taken debug events can occur regardless of the setting of MSR$_{DE}$.

When an Interrupt Taken debug event occurs, DBSR$_{IRPT}$ is set to 1 to record the debug exception. If MSR$_{DE}$=0, DBSR$_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR$_{DE}$=1 (i.e. Debug interrupts are enabled) at the time of the Interrupt Taken debug event, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and Critical Save/Restore Register 0/Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug] will be set to the address of the interrupt vector which caused the Interrupt Taken debug event. No instructions at the base interrupt handler will have been executed.

If MSR$_{DE}$=0 (i.e. Debug interrupts are disabled) at the time of the Interrupt Taken debug event, a Debug interrupt will not occur, and the handler for the interrupt which caused the Interrupt Taken debug event will be allowed to execute.
Later, if the debug exception has not been reset by clearing DBSR\_UDE, and MSR\_DE is set to 1, a delayed Debug interrupt will occur. In this case, CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR\_DE to 1. Software in the Debug interrupt handler can observe the DBSR\_IDE bit to determine how to interpret the value in CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug].

### 10.4.7 Return Debug Event

A Return debug event (RET) occurs if DBCR\_RET = 1 and an attempt is made to execute an rfi (and also rfgi <E.HV>). Return debug events can occur regardless of the setting of MSR\_DE.

When a Return debug event occurs, DBSR\_RET is set to 1 to record the debug exception. If MSR\_DE = 0, DBSR\_IDE is also set to 1 to record the imprecise debug event.

If MSR\_DE = 1 at the time of the Return Debug event, a Debug interrupt will occur immediately, and CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug] will be set to the address of the rfi.

If MSR\_DE = 0 at the time of the Return Debug event, a Debug interrupt will not occur.

Later, if the debug exception has not been reset by clearing DBSR\_RET, and MSR\_DE is set to 1, a delayed imprecise Debug interrupt will occur. In this case, CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR\_DE to 1. An imprecise Debug interrupt can be caused by executing an rfi when DBCR\_RET = 1 and MSR\_DE = 0, and the execution of that rfi happens to cause MSR\_DE to be set to 1. Software in the Debug interrupt handler can observe the DBSR\_IDE bit to determine how to interpret the value in CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug].

### 10.4.8 Unconditional Debug Event

An Unconditional debug event (UDE) occurs when the Unconditional Debug Event (UDE) signal is activated by the debug mechanism. The exact definition of the UDE signal and how it is activated is implementation-dependent. The Unconditional debug event is the only debug event which does not have a corresponding enable bit for the event in DBCR\_UDE (hence the name of the event). The Unconditional debug event can occur regardless of the setting of MSR\_DE.

When an Unconditional debug event occurs, the DBSR\_UDE bit is set to 1 to record the Debug exception. If MSR\_DE = 0, DBSR\_IDE is also set to 1 to record the imprecise debug event.

If MSR\_DE = 1 (i.e. Debug interrupts are enabled) at the time of the Unconditional Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR\_DE to 1. Software in the Debug interrupt handler can observe DBSR\_IDE to determine how to interpret the value in CSR\_R0/DSR\_R0 [Category: Embedded.Enhanced Debug].

### 10.4.9 Critical Interrupt Taken Debug Event

A Critical Interrupt Taken debug event (CIRPT) occurs if DBCR\_CIRPT = 1 (i.e. Critical Interrupt Taken debug events are enabled) and a critical interrupt occurs. A critical interrupt is any interrupt that saves state in CSR\_R0 and CSR\_R1 when the interrupt is taken. Critical Interrupt Taken debug events can occur regardless of the setting of MSR\_DE.

When a Critical Interrupt Taken debug event occurs, DBSR\_CIRPT is set to 1 to record the debug event. If MSR\_DE = 0, DBSR\_IDE is also set to 1 to record the imprecise debug event.

If MSR\_DE = 1 (i.e. Debug Interrupts are enabled) at the time of the Critical Interrupt Taken debug event, a Debug Interrupt will occur immediately (provided there is no higher priority exception which is enabled to cause an interrupt), and DSRR\_R0 will contain the address of the first instruction of the critical interrupt handler. No instructions at the critical interrupt handler will have been executed.

If MSR\_DE = 0 (i.e. Debug Interrupts are disabled) at the time of the Critical Interrupt Taken debug event, a Debug Interrupt will not occur, and the handler for the critical interrupt which caused the debug event will be allowed to execute normally. Later, if the debug exception has not been reset by clearing DBSR\_CIRPT and MSR\_DE is set to 1, a delayed Debug Interrupt will occur. In this case DSRR\_R0 will contain the address of the instruction after the one that set MSR\_DE = 1. Software in the Debug Interrupt handler can observe DBSR\_IDE to determine how to interpret the value in DSRR\_R0.
10.4.10 Critical Interrupt Return Debug Event [Category: Embedded Enhanced Debug]

A Critical Interrupt Return debug event (CRET) occurs if DBCR0CRET = 1 (i.e. Critical Interrupt Return debug events are enabled) and an attempt is made to execute an rfci instruction. Critical Interrupt Return debug events can occur regardless of the setting of MSRDE.

When a Critical Interrupt Return debug event occurs, DBSRCRET is set to 1 to record the debug event. If MSRDE=0, DBSRIDE is also set to 1 to record the imprecise debug event.

If MSRDE = 1 (i.e. Debug Interrupts are enabled) at the time of the Critical Interrupt Return debug event, a Debug Interrupt will occur immediately (provided there is no higher priority exception which is enabled to cause an interrupt), and DSRR0 will be set to the address of the rfci instruction.

If MSRDE = 0 (i.e. Debug Interrupts are disabled) at the time of the Critical Interrupt Return debug event, a Debug Interrupt will not occur. Later, if the debug exception has not been reset by clearing DBSRCRET and MSRDE is set to 1, a delayed Debug Interrupt will occur. In this case DSRR0 will contain the address of the instruction after the one that set MSRDE = 1. An imprecise Debug Interrupt can be caused by executing an rfci when DBCR0CRET = 1 and MSRDE = 0, and the execution of the rfci happens to cause MSRDE to be set to 1. Software in the Debug Interrupt handler can observe DBSRIDE to determine how to interpret the value in DSRR0.

10.5 Debug Registers

This section describes debug-related registers that are accessible to software. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

10.5.1 Debug Control Registers

Debug Control Register 0 (DBCR0), Debug Control Register 1 (DBCR1), and Debug Control Register 2 (DBCR2) are each 32-bit registers. Bits of DBCR0, DBCR1, and DBCR2 are numbered 32 (most-significant bit) to 63 (least-significant bit). DBCR0, DBCR1, and DBCR2 are used to enable debug events, reset the thread, control timer operation during debug events, and set the debug mode of the thread.

10.5.1.1 Debug Control Register 0 (DBCR0)

The contents of the DBCR0 can be read into bits 32:63 of register RT using mfspr RT, DBCR0, setting bits 0:31 of RT to 0. The contents of bits 32:63 of register RS can be written to the DBCR0 using mt spr DBCR0, RS. The bit definitions for DBCR0 are shown below.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>External Debug Mode (EDM)</td>
</tr>
<tr>
<td></td>
<td>The EDM bit is a read-only bit that reflects whether the thread is controlled by an external debug facility. When EDM is set, internal debug mode is suppressed and the taking of debug interrupts does not occur.</td>
</tr>
<tr>
<td>0</td>
<td>The thread is not in external debug mode.</td>
</tr>
<tr>
<td>1</td>
<td>The thread is in external debug mode.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtualized Implementation Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>In a virtualized implementation when EDM=1, the value of MSRDE is not specified and is not modifiable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>33</th>
<th>Internal Debug Mode (IDM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Debug interrupts are disabled.</td>
</tr>
<tr>
<td>1</td>
<td>If MSRDE=1, then the occurrence of a debug event or the recording of an earlier debug event in the Debug Status Register when MSRDE=0 or DBCR0IDM=0 will cause a Debug interrupt.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software must clear debug event status in the Debug Status Register in the Debug interrupt handler when a Debug interrupt is taken before re-enabling interrupts via MSRDE. Otherwise, redundant Debug interrupts will be taken for the same debug event.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>34:35</th>
<th>Reset (RST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No action</td>
</tr>
<tr>
<td>01</td>
<td>Implementation-specific</td>
</tr>
<tr>
<td>10</td>
<td>Implementation-specific</td>
</tr>
<tr>
<td>11</td>
<td>Implementation-specific</td>
</tr>
</tbody>
</table>

**Warning:** Writing 0b01, 0b10, or 0b11 to these bits may cause a thread reset to occur.

<table>
<thead>
<tr>
<th>36</th>
<th>Instruction Completion Debug Event (ICMP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ICMP debug events are disabled</td>
</tr>
<tr>
<td>1</td>
<td>ICMP debug events are enabled</td>
</tr>
</tbody>
</table>

**Note:** Instruction Completion will not cause an ICMP debug event if MSRDE=0.

<table>
<thead>
<tr>
<th>37</th>
<th>Branch Taken Debug Event Enable (BRT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BRT debug events are disabled</td>
</tr>
<tr>
<td>1</td>
<td>BRT debug events are enabled</td>
</tr>
</tbody>
</table>
**Note:** Taken branches will not cause a BRT debug event if MSR<sub>DE</sub>=0.

38 **Interrupt Taken Debug Event Enable** (IRPT)

- 0 IRPT debug events are disabled
- 1 IRPT debug events are enabled

**Note:** Critical interrupts will not cause an IRPT Debug event even if MSR<sub>DE</sub>=0. If the Embedded.Enhanced Debug category is supported, see Section 10.4.9.

39 **Trap Debug Event Enable** (TRAP)

- 0 TRAP debug events cannot occur
- 1 TRAP debug events can occur

40 **Instruction Address Compare 1 Debug Event Enable** (IAC1)

- 0 IAC1 debug events cannot occur
- 1 IAC1 debug events can occur

41 **Instruction Address Compare 2 Debug Event Enable** (IAC2)

- 0 IAC2 debug events cannot occur
- 1 IAC2 debug events can occur

42 **Instruction Address Compare 3 Debug Event Enable** (IAC3)

- 0 IAC3 debug events cannot occur
- 1 IAC3 debug events can occur

43 **Instruction Address Compare 4 Debug Event Enable** (IAC4)

- 0 IAC4 debug events cannot occur
- 1 IAC4 debug events can occur

44:45 **Data Address Compare 1 Debug Event Enable** (DAC1)

- 00 DAC1 debug events cannot occur
- 01 DAC1 debug events can occur only if a store-type data storage access
- 10 DAC1 debug events can occur only if a load-type data storage access
- 11 DAC1 debug events can occur on any data storage access

46:47 **Data Address Compare 2 Debug Event Enable** (DAC2)

- 00 DAC2 debug events cannot occur
- 01 DAC2 debug events can occur only if a store-type data storage access
- 10 DAC2 debug events can occur only if a load-type data storage access
- 11 DAC2 debug events can occur on any data storage access

48 **Return Debug Event Enable** (RET)

- 0 RET debug events cannot occur
- 1 RET debug events can occur

**Note:** Return From Critical Interrupt will not cause an RET debug event if MSR<sub>DE</sub>=0. If the Embedded.Enhanced Debug category is supported, see Section 10.4.10

49:56 Reserved

57 **Critical Interrupt Taken Debug Event** (CIRPT) [Category: Embedded.Enhanced Debug]

A Critical Interrupt Taken Debug Event occurs when DBCR0<sub>CIRPT</sub> = 1 and a critical interrupt (any interrupt that uses the critical class, i.e. uses CSRR0 and CSRR1) occurs.

- 0 Critical interrupt taken debug events are disabled.
- 1 Critical interrupt taken debug events are enabled.

58 **Critical Interrupt Return Debug Event** (CRET) [Category: Embedded.Enhanced Debug]

A Critical Interrupt Return Debug Event occurs when DBCR0<sub>CRET</sub> = 1 and a return from critical interrupt (an rfc<sub>i</sub> instruction is executed) occurs.

- 0 Critical interrupt return debug events are disabled.
- 1 Critical interrupt return debug events are enabled.

59:62 Implementation-dependent

63 **Freeze Timers on Debug Event** (FT)

- 0 Enable clocking of timers and Time Base
- 1 Disable clocking of timers and Time Base if any DBSR bit is set (except MRR)

**Virtualized Implementation Note**

The FT bit may not be supported in virtualized implementations.

This register is hypervisor privileged.

10.5.1.2 Debug Control Register 1 (DBCR1)

The contents of the DBCR1 can be read into bits 32:63 a register RT using **mtspr** RT, DBCR1, setting bits 0:31 of RT to 0. The contents of bits 32:63 of register RS can be written to the DBCR1 using **mtspr** DBCR1, RS. The bit definitions for DBCR1 are shown below.

**Bit(s) Description**

32:33 **Instruction Address Compare 1 User/ Supervisor Mode** (IAC1US)

- 00 IAC1 debug events can occur
- 01 Reserved
10 IAC1 debug events can occur only if MSRPR=0
11 IAC1 debug events can occur only if MSRPR=1

34:35 Instruction Address Compare 1 Effective/Real Mode (IAC1ER)
00 IAC1 debug events are based on effective addresses
01 IAC1 debug events are based on real addresses
10 IAC1 debug events are based on effective addresses and can occur only if MSRS=0
11 IAC1 debug events are based on effective addresses and can occur only if MSRS=1

36:37 Instruction Address Compare 2 User/Supervisor Mode (IAC2US)
00 IAC2 debug events can occur
01 Reserved
10 IAC2 debug events can occur only if MSRPR=0
11 IAC2 debug events can occur only if MSRPR=1

38:39 Instruction Address Compare 2 Effective/Real Mode (IAC2ER)
00 IAC2 debug events are based on effective addresses
01 IAC2 debug events are based on real addresses
10 IAC2 debug events are based on effective addresses and can occur only if MSRS=0
11 IAC2 debug events are based on effective addresses and can occur only if MSRS=1

40:41 Instruction Address Compare 1/2 Mode (IAC12M)
00 Exact address compare
IAC1 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC1.
IAC2 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC2.

01 Address bit match
IAC1 and IAC2 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, also ANDed with the contents of IAC2.
If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

10 Inclusive address range compare
IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2.
If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

11 Exclusive address range compare
IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2.
If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

42:47 Reserved

48:49 Instruction Address Compare 3 User/Supervisor Mode (IAC3US)
00 IAC3 debug events can occur
01 Reserved
10 IAC3 debug events can occur only if MSRPR=0
11 IAC3 debug events can occur only if MSRPR=1

50:51 Instruction Address Compare 3 Effective/Real Mode (IAC3ER)
00 IAC3 debug events are based on effective addresses
01 IAC3 debug events are based on real addresses
10 IAC3 debug events are based on effective addresses and can occur only if MSRS=0
11 IAC3 debug events are based on effective addresses and can occur only if MSRS=1

52:53 Instruction Address Compare 4 User/Supervisor Mode (IAC4US)
00 IAC4 debug events can occur
01 Reserved
10 IAC4 debug events can occur only if MSRPR=0
11 IAC4 debug events can occur only if MSRPR=1

54:55 Instruction Address Compare 4 Effective/Real Mode (IAC4ER)
00 IAC4 debug events are based on effective addresses
01 IAC4 debug events are based on real addresses
10 IAC4 debug events are based on effective addresses and can occur only if MSRS=0
11 IAC4 debug events are based on effective addresses and can occur only if MSRS=1
56:57 Instruction Address Compare 3/4 Mode (IAC34M)

00 Exact address compare
IAC3 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC3.
IAC4 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC4.

01 Address bit match
IAC3 and IAC4 debug events can occur only if the address of the data storage access, ANDed with the contents of IAC4 are equal to the contents of IAC3, also ANDed with the contents of IAC4.
If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

10 Inclusive address range compare
IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4.
If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

11 Exclusive address range compare
IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4.
If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

58:63 Reserved
This register is hypervisor privileged.

10.5.1.3 Debug Control Register 2 (DBCR2)
The contents of the DBCR2 can be copied into bits 32:63 register RT using mfspr RT,DBCR2, setting bits 0:31 of register RT to 0. The contents of bits 32:63 of a register RS can be written to the DBCR2 using mtspr DBCR2,RS. The bit definitions for DBCR2 are shown below.

Bit(s) Description
32:33 Data Address Compare 1 User/Supervisor Mode (DAC1US)

00 DAC1 debug events can occur
01 Reserved
10 DAC1 debug events can occur only if MSRPR=0
11 DAC1 debug events can occur only if MSRPR=1

34:35 Data Address Compare 1 Effective/Real Mode (DAC1ER)

00 DAC1 debug events are based on effective addresses
01 DAC1 debug events are based on real addresses
10 DAC1 debug events are based on effective addresses and can occur only if MSRDS=0
11 DAC1 debug events are based on effective addresses and can occur only if MSRDS=1

36:37 Data Address Compare 2 User/Supervisor Mode (DAC2US)

00 DAC2 debug events can occur
01 Reserved
10 DAC2 debug events can occur only if MSRPR=0
11 DAC2 debug events can occur only if MSRPR=1

38:39 Data Address Compare 2 Effective/Real Mode (DAC2ER)

00 DAC2 debug events are based on effective addresses
01 DAC2 debug events are based on real addresses
10 DAC2 debug events are based on effective addresses and can occur only if MSRDS=0
11 DAC2 debug events are based on effective addresses and can occur only if MSRDS=1

40:41 Data Address Compare 1/2 Mode (DAC12M)

00 Exact address compare

Architecture Note

For IAC12M inclusive (10) and exclusive (11) range comparisons, either DBCR0IAC1 or DBCR0IAC2 can be set to one in order to enable the range comparison. It is permissible for both DBCR0IAC1 and DBCR0IAC2 to be enabled. In that case, both DBSR1IAC1 and DBSR1IAC2 will be set for a range match.
The same behavior holds for IAC34M with the appropriate substitutions.
DAC1 debug events can occur only if the address of the data storage access is equal to the value specified in DAC1.

DAC2 debug events can occur only if the address of the data storage access is equal to the value specified in DAC2.

01 Address bit match

DAC1 and DAC2 debug events can occur only if the address of the data storage access, ANDed with the contents of DAC2 are equal to the contents of DAC1, also ANDed with the contents of DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

10 Inclusive address range compare

DAC1 and DAC2 debug events can occur only if the address of the data storage access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

11 Exclusive address range compare

DAC1 and DAC2 debug events can occur only if the address of the data storage access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

42:63 Reserved

Architecture Note

Bits 44:63 will be among the last to be assigned a meaning as these bits contained the DVC (Data Value Compare) function in earlier versions of the architecture.

This register is hypervisor privileged.

Architecture Note

For DAC12M inclusive (10) and exclusive (11) range comparisons, either DBCR0DAC1 or DBCR0DAC2 can be set to one in order to enable the range comparison. It is permissible for both DBCR0DAC1 and DBCR0DAC2 to be enabled. In that case, both DBSRDAC1 and DBSRDAC2 will be set for a range match.

The same behavior holds for DAC34M with the appropriate substitutions.

10.5.2 Debug Status Register

The Debug Status Register (DBSR) is a 32-bit register and contains status on debug events and the most recent thread reset.

The DBSR is set via hardware, and read and cleared via software. The contents of the DBSR can be read into bits 32:63 of a register RT using the\texttt{mfspr} instruction, setting bits 0:31 of RT to zero. Bits in the DBSR can be cleared using the\texttt{mtspr} instruction. Clearing is done by writing bits 32:63 of a register to the DBSR with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the DBSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The bit definitions for the DBSR are shown below:

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Imprecise Debug Event (IDE)</td>
</tr>
<tr>
<td></td>
<td>Set to 1 if MSR\textsubscript{DE}=0 and a debug event causes its respective Debug Status Register bit to be set to 1.</td>
</tr>
<tr>
<td>33</td>
<td>Unconditional Debug Event (UDE)</td>
</tr>
<tr>
<td></td>
<td>Set to 1 if an Unconditional debug event occurred. See Section 10.4.8.</td>
</tr>
<tr>
<td>34:35</td>
<td>Most Recent Reset (MRR)</td>
</tr>
<tr>
<td></td>
<td>Set to one of three values when a reset occurs. These two bits are undefined at power-up.</td>
</tr>
<tr>
<td></td>
<td>00 No reset occurred since these bits last cleared by software</td>
</tr>
<tr>
<td></td>
<td>01 Implementation-dependent reset information</td>
</tr>
<tr>
<td></td>
<td>10 Implementation-dependent reset information</td>
</tr>
<tr>
<td></td>
<td>11 Implementation-dependent reset information</td>
</tr>
<tr>
<td>36</td>
<td>Instruction Complete Debug Event (ICMP)</td>
</tr>
<tr>
<td></td>
<td>Set to 1 if an Instruction Completion debug event occurred and DBCR0\textsubscript{ICMP}=1. See Section 10.4.5.</td>
</tr>
<tr>
<td>37</td>
<td>Branch Taken Debug Event (BRT)</td>
</tr>
<tr>
<td></td>
<td>Set to 1 if a Branch Taken debug event occurred and DBCR0\textsubscript{BRT}=1. See Section 10.4.4.</td>
</tr>
<tr>
<td>38</td>
<td>Interrupt Taken Debug Event (IRPT)</td>
</tr>
<tr>
<td></td>
<td>Set to 1 if an Interrupt Taken debug event occurred and DBCR0\textsubscript{IRPT}=1. See Section 10.4.6.</td>
</tr>
<tr>
<td>39</td>
<td>Trap Instruction Debug Event (TRAP)</td>
</tr>
</tbody>
</table>

Set to 1 if a Trap Instruction debug event occurred and DBCR0\text{TRAP}=1. See Section 10.4.3.

**Instruction Address Compare 1 Debug Event** (IAC1)
Set to 1 if an IAC1 debug event occurred and DBCR0\text{IAC1}=1. See Section 10.4.1.

**Instruction Address Compare 2 Debug Event** (IAC2)
Set to 1 if an IAC2 debug event occurred and DBCR0\text{IAC2}=1. See Section 10.4.1.

**Instruction Address Compare 3 Debug Event** (IAC3)
Set to 1 if an IAC3 debug event occurred and DBCR0\text{IAC3}=1. See Section 10.4.1.

**Instruction Address Compare 4 Debug Event** (IAC4)
Set to 1 if an IAC4 debug event occurred and DBCR0\text{IAC4}=1. See Section 10.4.1.

**Data Address Compare 1 Read Debug Event** (DAC1R)
Set to 1 if a read-type DAC1 debug event occurred and DBCR0\text{DAC1}=0b10 or DBCR0\text{DAC1}=0b11. See Section 10.4.2.

**Data Address Compare 1 Write Debug Event** (DAC1W)
Set to 1 if a write-type DAC1 debug event occurred and DBCR0\text{DAC1}=0b01 or DBCR0\text{DAC1}=0b11. See Section 10.4.2.

**Data Address Compare 2 Read Debug Event** (DAC2R)
Set to 1 if a read-type DAC2 debug event occurred and DBCR0\text{DAC2}=0b10 or DBCR0\text{DAC2}=0b11. See Section 10.4.2.

**Data Address Compare 2 Write Debug Event** (DAC2W)
Set to 1 if a write-type DAC2 debug event occurred and DBCR0\text{DAC2}=0b01 or DBCR0\text{DAC2}=0b11. See Section 10.4.2.

**Return Debug Event** (RET)
Set to 1 if a Return debug event occurred and DBCR0\text{RET}=1. See Section 10.4.2.

0 Critical interrupt taken debug events are disabled.
1 Critical interrupt taken debug events are enabled.

**Critical Interrupt Return Debug Event** (CRET) [Category: Embedded.Enhanced Debug]
A Critical Interrupt Return Debug Event occurs when DBCR0CRET=1 and a return from critical interrupt (an \textit{rfci} instruction is executed) occurs.

0 Critical interrupt return debug events are disabled.
1 Critical interrupt return debug events are enabled.

Implementation-dependent
This register is hypervisor privileged.
10.5.3 Debug Status Register Write Register (DBSRWR)

The Debug Status Register Write Register (DBSRWR) allows a hypervisor state program to write the contents of the Debug Status Register (see Section 10.5.2, “Debug Status Register”). The format of the DBSRWR is shown in Figure 86 below.

![DBSRWR Diagram](image)

Figure 86. Debug Status Register Write Register

The DBSRWR is provided as a means to restore the contents of the DBSR on a partition switch.

The DBSRWR is hypervisor privileged.

Writing DBSRWR changes the value in the DBSR. Writing non-zero bits may enable an imprecise Debug exception which may cause later imprecise Debug Interrupts. In order to correctly write DBSRWR, software should ensure that MSRDE = 0 when the value is written and perform a context synchronizing operation before setting MSRDE to 1.

10.5.4 Instruction Address Compare Registers

The Instruction Address Compare Register 1, 2, 3, and 4 (IAC1, IAC2, IAC3, and IAC4 respectively) are each 64-bits, with bit 63 being reserved.

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in either IAC1, IAC2, IAC3, or IAC4, inside or outside a range specified by IAC1 and IAC2 or, inside or outside a range specified by IAC3 and IAC4, or to blocks of addresses specified by the combination of the IAC1 and IAC2, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address (see Section 10.4.1 on page 1215).

The contents of the Instruction Address Compare i Register (where $i = \{1, 2, 3, \text{ or } 4\}$) can be read into register RT using $\text{mfspr RT, IAC}i$. The contents of register RS can be written to the Instruction Address Compare i Register using $\text{mtspr IAC}i, RS$.

This register is hypervisor privileged.

10.5.5 Data Address Compare Registers

The Data Address Compare Register 1 and 2 (DAC1 and DAC2 respectively) are each 64-bits.

A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either the DAC1 or DAC2, inside or outside a range specified by the DAC1 and DAC2, or to blocks of addresses specified by the combination of the DAC1 and DAC2 (see Section 10.4.2).

The contents of the Data Address Compare i Register (where $i = \{1 \text{ or } 2\}$) can be read into register RT using $\text{mfspr RT, DAC}i$. The contents of register RS can be written to the Data Address Compare i Register using $\text{mtspr DAC}i, RS$.

The contents of the DAC1 or DAC2 are compared to the address generated by a data storage access instruction.

These registers are hypervisor privileged.
10.6 Debugger Notify Halt Instruction

The *dnh* instruction provides the means for the transfer of information between the thread and an implementation-dependent external debug facility. *dnh* also causes the thread to stop fetching and executing instructions.

**Debugger Notify Halt**  
**XFX-form**

\[
\text{dnh \ DUI,DUIS}
\]

if enabled by implementation-dependent means
then
  \[
  \text{implementation-dependent register} \leftarrow \text{DUI}
  \]
  halt thread
else
  illegal instruction exception

Execution of the *dnh* instruction causes the thread to stop fetching instructions and taking interrupts if execution of the instruction has been enabled. The contents of the DUI field are sent to the external debug facility to identify the reason for the halt.

If execution of the *dnh* instruction has not been previously enabled, executing the *dnh* instruction produces an Illegal Instruction exception. The means by which execution of the *dnh* instruction is enabled is implementation-dependent.

The current state of the debug facility, whether the thread is in IDM or EDM mode has no effect on the execution of the *dnh* instruction.

The instruction is context synchronizing.

**Programming Note**

The DUI field in the instruction may be used to pass information to an external debug facility. After the *dnh* instruction has executed, the instruction itself can be read back by the Illegal Instruction Interrupt handler or the external debug facility if the contents of the DUI field are of interest. If the thread entered the Illegal Instruction Interrupt handler, software can use SRR0 to obtain the address of the *dnh* instruction which caused the handler to be invoked.

**Special Registers Altered:**

None
Chapter 11. Processor Control [Category: Embedded.Processor Control]

11.1 Overview

The Processor Control facility provides a mechanism for threads within a coherence domain to send messages to all devices in the coherence domain. The facility provides a mechanism for sending interrupts that are not dependent on the interrupt controller to threads and allows message filtering by the threads that receive the message.

The Processor Control facility is also useful for sending messages to a device that provides specialized services such as secure boot operations controlled by a security device.

The Processor Control facility defines how threads send messages and what actions threads take on the receipt of a message. The actions taken by devices other than threads are not defined.

Programming Note

A common use of `msgsnd` is to deliver an external interrupt to a partition which has set MSR_{EE}=0. A guest doorbell message will interrupt to the hypervisor when MSR_{EE}=1 and MSR_{GS}=1. The hypervisor can then deliver the external interrupt to the partition. For Servers, a similar set of operations can also be performed using the Book III-S `msgsnd` instruction. These operations achieve a result analogous to the mediated external interrupt in Book III-S.

11.2 Programming Model

Threads initiate a message by executing the `msgsnd` instruction and specifying a message type and message payload in a general purpose register. Sending a message causes the message to be sent to all the devices, including the sending thread, in the coherence domain in a reliable manner.

Each device receives all messages that are sent. The actions that a device takes are dependent on the message type and payload. There are no restrictions on what messages a thread can send.

To provide inter thread interrupt capability the following do<e:hi>rbell</e:hi> message types are defined:

- Processor Doorbell
- Processor Doorbell Critical
- Guest Processor Doorbell <E.HV>
- Guest Processor Doorbell Critical <E.HV>
- Guest Processor Doorbell Machine Check <E.HV>

A doorbell message causes an interrupt to occur on threads when the message is received and the thread determines through examination of the payload that the message should be accepted. The examination of the payload for this purpose is termed filtering. The acceptance of a doorbell message causes an exception to be generated on the accepting thread.

Threads accept and filter messages defined in Section 11.2.1. Threads may also accept other implementation-dependent defined messages.

11.2.1 Message Handling and Filtering

Threads filter, accept, and handle message types defined as follows. The message type is specified in the message and is determined by the contents of register RB_{32:36} used as the operand in the `msgsnd` instruction. The message type is interpreted as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Doorbell Interrupt</strong> (DBELL)</td>
</tr>
<tr>
<td></td>
<td>A Processor Doorbell exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Processor Doorbell Interrupt occurs when no higher priority exception exists, a Processor Doorbell exception exists, and the interrupt is enabled (MSR_{EE}=1). If Category: Embedded.Hypervisor is supported, the interrupt is enabled if (MSR_{EE}=1 or MSR_{GS}=1).</td>
</tr>
</tbody>
</table>
1. **Doorbell Critical Interrupt** (DBELL_CRIT)  
A Processor Doorbell Critical exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Processor Doorbell Critical exception exists, and the interrupt is enabled (MSR\text{CE}=1). If Category: Embedded.Hypervisor is supported, the interrupt is enabled if (MSR\text{CE}=1 or MSR\text{GS}=1).

2. **Guest Doorbell Interrupt** (G_DBELL) 
   
   A Guest Processor Doorbell exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Guest Processor Doorbell Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell exception exists, and the interrupt is enabled (MSR\text{EE}=1 and MSR\text{GS}=1).

3. **Guest Doorbell Interrupt Critical** (G_DBELL_CRIT) 
   
   A Guest Processor Doorbell Critical exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Guest Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Critical exception exists, and the interrupt is enabled (MSR\text{CE}=1 and MSR\text{GS}=1).

4. **Guest Doorbell Interrupt Machine Check** (G_DBELL_MC) 
   
   A Guest Processor Doorbell Machine Check exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Guest Processor Doorbell Machine Check Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Machine Check exception exists, and the interrupt is enabled (MSR\text{ME}=1 and MSR\text{GS}=1).

Message types other than these and their associated actions are implementation-dependent.

**11.2.2 Doorbell Message Filtering**

A thread receiving a DBELL message will filter the message and either ignore the message or accept the message and generate a Processor Doorbell exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB\text{37:63} used as the operand in the `msgsnd` instruction. The payload bits are defined below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Broadcast (BRDCAST)</td>
</tr>
<tr>
<td></td>
<td>If set, the message is accepted by all threads regardless of the value of the PIR register and the value of PIRTAG.</td>
</tr>
<tr>
<td></td>
<td>0 If the value of PIR and PIRTAG are equal a Processor Doorbell exception is generated.</td>
</tr>
<tr>
<td></td>
<td>1 A Processor Doorbell exception is generated regardless of the value of PIRTAG and PIR.</td>
</tr>
<tr>
<td>38:49</td>
<td>LPID Tag (LPIDTAG)</td>
</tr>
<tr>
<td></td>
<td>The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.</td>
</tr>
<tr>
<td>50:63</td>
<td>PIR Tag (PIRTAG)</td>
</tr>
<tr>
<td></td>
<td>The contents of this field are compared with bits 50:63 of the PIR register.</td>
</tr>
</tbody>
</table>

If category E.HV is supported by the thread on which a DBELL message is received, the message will only be accepted if it is for this partition (payload\text{LPIDTAG} = LPIDR) or it is for all partitions (payload\text{LPIDTAG} = 0) and it meets the additional criteria for acceptance below.

If a DBELL message is received by a thread, the message is accepted and a Processor Doorbell exception is generated if one of the following conditions exist:

- This is a broadcast message (payload\text{BRDCAST}=1);
- The message is intended for this thread (PIR\text{50:63}=payload\text{PIRTAG});

The exception condition remains until a Processor Doorbell Interrupt is taken, or a `msgclr` instruction is executed on the receiving thread with a message type of DBELL. A change to any of the filtering criteria (i.e. changing the PIR register) will not clear a pending Processor Doorbell exception.

DBELL messages are not cumulative. That is, if a DBELL message is accepted and the interrupt is pended because MSR\text{EE}=0, additional DBELL messages that would be accepted are ignored until the Processor Doorbell exception is cleared by taking the interrupt or cleared by executing a `msgclr` with a message type of DBELL on the receiving thread.

The temporal relationship between when a DBELL message is sent and when it is received in a given thread is not defined.

**11.2.2.1 Doorbell Critical Message Filtering**

A thread receiving a DBELL_CRIT message type will filter the message and either ignore the message or
accept the message and generate a Processor Doorbell Critical exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB\textsubscript{37:63} used as the operand in the \texttt{msgsnd} instruction. The payload bits are defined below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Broadcast (BRDCAST)</td>
</tr>
<tr>
<td>38:49</td>
<td>\texttt{LPID Tag} (LPIDTAG)</td>
</tr>
<tr>
<td>50:63</td>
<td>\texttt{PIR Tag} (PIRTAG)</td>
</tr>
</tbody>
</table>

If category E.HV is supported by the thread on which a DBELL\_CRIT message is received, the message will only be accepted if it is for this partition (payload\textsubscript{LPIDTAG} = LPIDR) or it is for all partitions (payload\textsubscript{LPIDTAG} = 0) and it meets the additional criteria for acceptance below.

If a DBELL\_CRIT message is received by a thread, the message is accepted and a Processor Doorbell Critical exception is generated if one of the following conditions exist:

- This is a broadcast message (payload\textsubscript{BRDCAST}=1);
- The message is intended for this thread (PIR\textsubscript{50:63}=payload\textsubscript{PIRTAG}).

DBELL\_CRIT messages are not cumulative. That is, if a DBELL\_CRIT message is accepted and the interrupt is pended because MSR\textsubscript{CE}=0, additional DBELL\_CRIT messages that would be accepted are ignored until the Processor Doorbell Critical exception is cleared by taking the interrupt or cleared by executing a \texttt{msgclr} with a message type of DBELL\_CRIT on the receiving thread.

The temporal relationship between when a DBELL\_CRIT message is sent and when it is received in a given thread is not defined.

The temporal relationship between when a DBELL\_CRIT message is sent and when it is received in a given thread is not defined.

11.2.2.2 Guest Doorbell Message Filtering [Category: Embedded.Hypervisor]

A thread receiving a G\_DBELL message type will filter the message and either ignore the message or accept the message and generate a Guest Processor Doorbell Critical exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB\textsubscript{37:63} used as the operand in the \texttt{msgsnd} instruction. The payload bits are defined below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Broadcast (BRDCAST)</td>
</tr>
<tr>
<td>38:49</td>
<td>\texttt{LPID Tag} (LPIDTAG)</td>
</tr>
<tr>
<td>50:63</td>
<td>\texttt{PIR Tag} (PIRTAG)</td>
</tr>
</tbody>
</table>

When a G\_DBELL message is received by a thread, the message will only be accepted if it is for this partition (payload\textsubscript{LPIDTAG} = LPIDR) or it is for all partitions (payload\textsubscript{LPIDTAG} = 0) and it meets the additional criteria for acceptance below.

The message is accepted and a Guest Processor Doorbell exception is generated if one of the following conditions exist:

- This is a broadcast message (payload\textsubscript{BRDCAST}=1);
- The message is intended for this thread (GPIR\textsubscript{50:63}=payload\textsubscript{PIRTAG}).

G\_DBELL messages are not cumulative. That is, if a G\_DBELL message is accepted and the interrupt is pended because MSR\textsubscript{CE}=0, additional G\_DBELL messages that would be accepted are ignored until the Guest Processor Doorbell exception is cleared by taking the interrupt or cleared by executing a \texttt{msgclr} with a message type of G\_DBELL on the receiving thread.

The temporal relationship between when a G\_DBELL message is sent and when it is received in a given thread is not defined.

The temporal relationship between when a G\_DBELL message is sent and when it is received in a given thread is not defined.
11.2.2.3 Guest Doorbell Critical Message Filtering [Category: Embedded.Hypervisor]

A thread receiving a G_DBELL_CRIT message type will filter the message and either ignore the message or accept the message and generate a Guest Processor Doorbell Critical exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB\textsubscript{37:63} used as the operand in the \texttt{msgsnd} instruction. The payload bits are defined below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Broadcast (BRDCAST)</td>
</tr>
<tr>
<td>0</td>
<td>If the value of GPIR and PIRTAG are equal a Guest Processor Doorbell Critical exception is generated.</td>
</tr>
<tr>
<td>1</td>
<td>A Guest Processor Doorbell Critical exception is generated regardless of the value of PIRTAG and GPIR.</td>
</tr>
<tr>
<td>38:49</td>
<td>LPID Tag (LPIDTAG)</td>
</tr>
<tr>
<td>50:63</td>
<td>PIR Tag (PIRTAG)</td>
</tr>
</tbody>
</table>

When a G_DBELL_CRIT message is received by a thread, the message will only be accepted if it is for this partition (payload\textsubscript{LPIDTAG} = LPIDR) or it is for all partitions (payload\textsubscript{LPIDTAG} = 0) and it meets the additional criteria for acceptance below.

If a G_DBELL_CRIT message is received by a thread, the message is accepted and a Guest Processor Doorbell Critical exception is generated if one of the following conditions exist:
- This is a broadcast message (payload\textsubscript{BRDCAST}=1);
- The message is intended for this thread (GPIR\textsubscript{50:63}=payload\textsubscript{PIRTAG});

G_DBELL_CRIT messages are not cumulative. That is, if a G_DBELL_CRIT message is accepted and the interrupt is pended because MSR\textsubscript{CE}=0, additional G_DBELL messages that would be accepted are ignored until the Guest Processor Doorbell Critical exception is cleared by taking the interrupt or cleared by executing a \texttt{msgclr} with a message type of G_DBELL_CRIT on the receiving thread.

The temporal relationship between when a G_DBELL_CRIT message is sent and when it is received in a given thread is not defined.

11.2.2.4 Guest Doorbell Machine Check Message Filtering [Category: Embedded.Hypervisor]

A thread receiving a G_DBELL_MC message type will filter the message and either ignore the message or accept the message and generate a Guest Processor Doorbell Machine Check exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB\textsubscript{37:63} used as the operand in the \texttt{msgsnd} instruction. The payload bits are defined below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Broadcast (BRDCAST)</td>
</tr>
<tr>
<td>0</td>
<td>If the value of GPIR and PIRTAG are equal a Guest Processor Doorbell Machine Check exception is generated.</td>
</tr>
<tr>
<td>1</td>
<td>A Guest Processor Doorbell Machine Check exception is generated regardless of the value of PIRTAG and GPIR.</td>
</tr>
<tr>
<td>38:49</td>
<td>LPID Tag (LPIDTAG)</td>
</tr>
<tr>
<td>50:63</td>
<td>PIR Tag (PIRTAG)</td>
</tr>
</tbody>
</table>

When a G_DBELL_MC message is received by a thread, the message will only be accepted if it is for this partition (payload\textsubscript{LPIDTAG} = LPIDR) or it is for all partitions (payload\textsubscript{LPIDTAG} = 0) and it meets the additional criteria for acceptance below.

If a G_DBELL_MC message is received by a thread, the message is accepted and a Guest Processor Doorbell Machine Check exception is generated if one of the following conditions exist:
- This is a broadcast message (payload\textsubscript{BRDCAST}=1);
- The message is intended for this thread (GPIR\textsubscript{50:63}=payload\textsubscript{PIRTAG});

G_DBELL_MC messages are not cumulative. That is, if a G_DBELL_MC message is accepted and the interrupt is pended because MSR\textsubscript{CE}=0, additional G_DBELL_MC messages that would be accepted are ignored until the Guest Processor Doorbell Machine Check exception is cleared by taking the interrupt or cleared by executing a \texttt{msgclr} with a message type of G_DBELL_MC on the receiving thread.

The temporal relationship between when a G_DBELL_MC message is sent and when it is received in a given thread is not defined.
## 11.3 Processor Control Instructions

`msgsnd` and `msgclr` instructions are provided for sending and clearing messages to threads and other devices in the coherence domain. These instructions are hypervisor privileged.

### Message Send

* `msgsnd` RB

<table>
<thead>
<tr>
<th>msgtype</th>
<th>RB</th>
<th>payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR(RB)32:36</td>
<td>GPR(RB)37:63</td>
<td></td>
</tr>
</tbody>
</table>

`msgsnd` sends a message to all devices in the coherence domain. The message contains a type and a payload. The message type (`msgtype`) is defined by the contents of RB32:36 and the message payload is defined by the contents of RB37:63. Message delivery is reliable and guaranteed. Each device may perform specific actions based on the message type and payload or may ignore messages. Consult implementation documentation for specific actions taken based on message type and payload.

For threads, actions taken on receipt of a message are defined in Section 11.2.1.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None

### Programming Note

If `msgsnd` is used to send a message notifying the receiver that updates have been made to storage, a `sync` instruction should be placed between the stores and the `msgsnd`. See Section 6.11.3, “Synchronize Instruction” on page 1124.

### Message Clear

* `msgclr` RB

<table>
<thead>
<tr>
<th>msgtype</th>
<th>RB</th>
<th>payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR(RB)32:36</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`msgclr` clears a message of `msgtype` previously accepted by the thread executing the `msgclr`. `msgtype` is defined by the contents of RB32:36. A message is said to be cleared when the associated interrupt is masked.

A context synchronizing instruction or event that is executed or occurs subsequent to the execution of `msgclr` ensures that the formerly pending exception will not result in an interrupt when the corresponding interrupt class is reenabled.

For threads, the types of messages that can be cleared are defined in Section 11.2.1.

This instruction is hypervisor privileged.

**Special Registers Altered:**

None

### Programming Note

Execution of a `msgclr` instruction that clears a pending exception when the associated interrupt is masked because the interrupt enable (MSR_{EE} or MSR_{CE}) is set to 0 will always clear the pending exception (and thus the interrupt will not occur) only if the instruction that sets MSR_{EE} or MSR_{CE} to 1 is context synchronizing or a context synchronizing operation occurs subsequent to the instruction but before MSR_{EE} or MSR_{CE} is set to 1.
Chapter 12. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of TLB entries, or the contents of other system resources that control the context in which a program executes can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing certain bits in the MSR has the side effect of changing how instruction addresses are calculated. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a context-altering instruction. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 13 (for data access) and Table 12 (for instruction fetch and execution).

The notation “CSI” in the tables means any context synchronizing instruction (e.g., sc, isync, rfi, rfci, rfmci, or rfdi [Category: Embedded. Enhanced Debug]). A context synchronizing interrupt (i.e., any interrupt except non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like “the synchronizing instruction”, below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, “the synchronizing instruction before (after) the context-altering instruction” should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as an rfi, rfgi [Category: Embedded.Hypervisor], rfci, rfmci, or rfdi [Category: Embedded. Enhanced Debug] that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing (e.g., rfi, etc.) or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 12, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.
Notes:

1. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect); it is also necessary to execute a `tlbsync` instruction.

2. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be

Table 12: Synchronization requirements for instruction fetch and/or execution

<table>
<thead>
<tr>
<th>Instruction or Event</th>
<th>Required Before</th>
<th>Required After</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfi</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rtc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfmci</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfdi[Category:E.ED]</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfgi</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr (GS)</td>
<td>none</td>
<td>CSI</td>
<td>2</td>
</tr>
<tr>
<td>mtmsr (LPIDR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (GIVPR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (DBSRWR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (GIVORI)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr (CM)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr (UCLE)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr (SPV)</td>
<td>none</td>
<td>CSI</td>
<td>4</td>
</tr>
<tr>
<td>mtmsr (CE)</td>
<td>none</td>
<td>none</td>
<td>4</td>
</tr>
<tr>
<td>mtmsr (PR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (FP)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (DE)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (ME)</td>
<td>none</td>
<td>CSI</td>
<td>3</td>
</tr>
<tr>
<td>mtmsr (FE0)</td>
<td>none</td>
<td>CSI</td>
<td>2</td>
</tr>
<tr>
<td>mtmsr (FE1)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (IS)</td>
<td>none</td>
<td>CSI</td>
<td>2</td>
</tr>
<tr>
<td>mtmsr (DEC)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtmsr (GDEC)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtmsr (PID)</td>
<td>none</td>
<td>CSI</td>
<td>2</td>
</tr>
<tr>
<td>mtmsr (VPR)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr (DBSR)</td>
<td>--</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>mtsp (DBC0,DBC1)</td>
<td>--</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>mtsp (IAC1,IAC2,IAC3, IAC4)</td>
<td>--</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>mtsp (IVORI)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsp (TSR)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (GTS)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (GTSWR)</td>
<td>none</td>
<td>CSI</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (TCR)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (GTCR)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (MMUCSR0)</td>
<td>none</td>
<td>CSI, or CSI and sync</td>
<td>6,9</td>
</tr>
<tr>
<td>TLB invalidate all</td>
<td>none</td>
<td>CSI and sync</td>
<td>6</td>
</tr>
<tr>
<td>mtsp (MCI)</td>
<td>none</td>
<td>CSI and sync</td>
<td>6</td>
</tr>
<tr>
<td>tlbivx</td>
<td>none</td>
<td>CSI, or CSI and sync</td>
<td>1,8</td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>CSI, or CSI and sync</td>
<td>1,8</td>
</tr>
<tr>
<td>tlbivax</td>
<td>none</td>
<td>CSI, or CSI and sync</td>
<td>1,6</td>
</tr>
</tbody>
</table>

Table 13: Synchronization requirements for data access

<table>
<thead>
<tr>
<th>Instruction or Event</th>
<th>Required Before</th>
<th>Required After</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfi</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rtc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfmci</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfdi[Category:E.ED]</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>rfgi</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtmsr (GS)</td>
<td>none</td>
<td>CSI</td>
<td>3</td>
</tr>
<tr>
<td>mtmsr (LPIDR)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (CM)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (PR)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (ME)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtmsr (DS)</td>
<td>none</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtsp (PID)</td>
<td>CSI</td>
<td>CSI</td>
<td></td>
</tr>
<tr>
<td>mtsp (GTS)</td>
<td>--</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>(DBC0,DBC1)</td>
<td>--</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>mtsp (IAC1,IAC2,IAC3, IAC4)</td>
<td>--</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>mtsp (IVORI)</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>mtsp (TSR)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (GTS)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (GTSWR)</td>
<td>none</td>
<td>CSI</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (TCR)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (GTCR)</td>
<td>none</td>
<td>none</td>
<td>7</td>
</tr>
<tr>
<td>mtsp (MMUCSR0)</td>
<td>CSI, or CSI and sync</td>
<td>6,9</td>
<td></td>
</tr>
<tr>
<td>TLB invalidate all</td>
<td>CSI and sync</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>mtsp (MCI)</td>
<td>CSI, or CSI and sync</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>tlbivx</td>
<td>CSI</td>
<td>CSI, or CSI and sync</td>
<td>1,6</td>
</tr>
<tr>
<td>Store(PTE)</td>
<td>none</td>
<td>CSI, or CSI and sync</td>
<td>1,8</td>
</tr>
<tr>
<td>tlbivax</td>
<td>none</td>
<td>CSI, or CSI and sync</td>
<td>1,6</td>
</tr>
</tbody>
</table>

Notes:

1. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect); it is also necessary to execute a `tlbsync` instruction.

2. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be
independent of whether the alteration has taken effect.

3. A context synchronizing instruction is required after altering MSRME to ensure that the alteration takes effect for subsequent Machine Check interrupts, which may not be recoverable and therefore may not be context synchronizing.

4. The effect of changing MSR_EE or MSR_CE is immediate.

If an mtmsr, wrtee, or wrteei instruction sets MSR_EE to '0', an External Input, DEC or FIT interrupt does not occur after the instruction is executed.

If an mtmsr, wrtee, or wrteei instruction changes MSR_EE from '0' to '1' when an External Input, Decrementer, Fixed-Interval Timer, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the mtmsr, wrtee, or wrteei is executed, and before the next instruction is executed in the program that set MSR_EE to '1'.

If an mtmsr instruction sets MSR_CE to '0', a Critical Input or Watchdog Timer interrupt does not occur after the instruction is executed.

If an mtmsr instruction changes MSR_CE from '0' to '1' when a Critical Input or Watchdog Timer interrupt occurs immediately after the mtmsr is executed, and before the next instruction is executed in the program that set MSR_CE to '1'.

5. Synchronization requirements for changing any of the Debug Facility Registers are implementation-dependent.

6. For data accesses, the context synchronizing instruction before the tlbwe, tlblix, or tlbivax instruction ensures that all storage accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause.

The context synchronizing instruction after the tlbwe, tlblix, or tlbivax ensures that subsequent storage accesses (data and instruction) will use the updated value in the TLB entry(s) being affected. It does not ensure that all storage accesses previously translated by the TLB entry(s) being updated have completed with respect to storage; if these completions must be assured, the tlbwe, tlblix, or tlbivax must be followed by a sync instruction as well as by a context synchronizing instruction.

7. The elapsed time between the Decrementer reaching zero, or the transition of the selected Time Base bit for the Fixed-Interval Timer or the Watchdog Timer, and the signalling of the Decrementer, Fixed-Interval Timer or the Watchdog Timer exception is not defined.

8. The notation "(sync, CSI)" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

Programming Note

The following sequence illustrates why it is necessary, for data accesses, to ensure that all storage accesses due to instructions before the tlbwe or tlbivax have completed to a point at which they have reported all exceptions they will cause. Assume that valid TLB entries exist for the target storage location when the sequence starts.

- A program issues a load or store to a page.
- The same program executes a tlbwe or tlbivax that invalidates the corresponding TLB entry.
- The Load or Store instruction finally executes, and gets a TLB Miss exception.
- The TLB Miss exception is semantically incorrect. In order to prevent it, a context synchronizing instruction must be executed between steps 1 and 2.

No software synchronization is required before the Store instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the Store instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the Store instruction will be performed using the old contents of the PTE.

The sync instruction after the Store instruction ensures that all lookups of the Page Table that are performed after the sync instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the sync instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).
The `sync` instruction also ensures that all storage accesses associated with instructions preceding the `sync` instruction, before the `sync` instruction is executed, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required or Alternate Coherence Mode attributes, before any data accesses caused by instructions following the `sync` instruction are performed with respect to that thread or mechanism.

9. After executing a `mtspr` that sets one of the TLB invalidate all bits in the MMUCSR0 to a 1, software must read MMUCSR0 using a `mfspr` instruction until the corresponding bit is zero and then perform the CSI, or CSI and `sync` as indicated in the “Required After” column.
Appendix A. Implementation-Dependent Instructions

This appendix documents architectural resources that are allocated for specific implementation-sensitive functions which have scope-limited utility. Implementations may exercise reasonable flexibility in implementing these functions, but that flexibility should be limited to that allowed in this appendix.

A.1 Embedded Cache Initialization [Category: Embedded.Cache Initialization]

**Data Cache Invalidate**

<table>
<thead>
<tr>
<th>X-form</th>
<th>Instruction Cache Invalidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>dci CT</td>
<td>ici CT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31 /</th>
<th>CT</th>
<th>///</th>
<th>///</th>
<th>454</th>
<th>/</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>dci</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If CT is not supported by the implementation, this instruction designates the primary data cache as the target data cache.

If CT is supported by the implementation, let CT designate either the primary data cache or another level of the data cache hierarchy, as specified in Section 4.3, “Cache Management Instructions”, in Book II, as the target data cache.

The contents of the target data cache of the thread executing the dci instruction are invalidated.

Software must place a sync instruction before the dci to guarantee all previous data storage accesses complete before the dci is performed.

Software must place a sync instruction after the dci to guarantee that the dci completes before any subsequent data storage accesses are performed.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

**Extended Mnemonics:**

Extended mnemonic for Data Cache Invalidate

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dccci</td>
<td>dci 0</td>
</tr>
</tbody>
</table>

**Instruction Cache Invalidate**

<table>
<thead>
<tr>
<th>X-form</th>
<th>Instruction Cache Invalidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ici CT</td>
<td>ici CT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31 /</th>
<th>CT</th>
<th>///</th>
<th>///</th>
<th>966</th>
<th>/</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>ici</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If CT is not supported by the implementation, this instruction designates the primary instruction cache as the target instruction cache.

If CT is supported by the implementation, let CT designate either the primary instruction cache or another level of the instruction cache hierarchy, as specified in Section 4.3, “Cache Management Instructions”, in Book II, as the target instruction cache.

The contents of the target instruction cache of the thread executing the ici instruction are invalidated.

Software must place a sync instruction before the ici to guarantee all previous instruction storage accesses complete before the ici is performed.

Software must place an isync instruction after the ici to invalidate any instructions that may have already been fetched from the previous contents of the instruction cache after the isync.

This instruction is hypervisor privileged.

**Special Registers Altered:**
None

**Extended Mnemonics:**

Extended mnemonic for Instruction Cache Invalidate

<table>
<thead>
<tr>
<th>Extended:</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>iccci</td>
<td>ici 0</td>
</tr>
</tbody>
</table>
A.2 Embedded Cache Debug Facility
[Category: Embedded.Cache Debug]

A.2.1 Embedded Cache Debug Registers

A.2.1.1 Data Cache Debug Tag Register High
The Data Cache Debug Tag Register High (DCDBTRH) is a 32-bit Special Purpose Register. The Data Cache Debug Tag Register High is read using `mfspr` and is set by `dcread`.

![DCDBTRH](image)

Figure 87. Data Cache Debug Tag Register High

### Programming Note
An example implementation of DCDBTRH could have the following content and format.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:55</td>
<td><strong>Tag Real Address</strong> (TRA)</td>
</tr>
<tr>
<td></td>
<td>Bits 0:23 of the lower 32 bits of the 36-bit real address associated with this cache block</td>
</tr>
<tr>
<td>56</td>
<td><strong>Valid</strong> (V)</td>
</tr>
<tr>
<td></td>
<td>The valid indicator for the cache block (1 indicates valid)</td>
</tr>
<tr>
<td>57:59</td>
<td>Reserved</td>
</tr>
<tr>
<td>60:63</td>
<td><strong>Tag Extended Real Address</strong> (TERA)</td>
</tr>
<tr>
<td></td>
<td>Upper 4 bits of the 36-bit real address associated with this cache block</td>
</tr>
</tbody>
</table>

Implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.

A.2.1.2 Data Cache Debug Tag Register Low
The Data Cache Debug Tag Register Low (DCDBTRL) is a 32-bit Special Purpose Register. The Data Cache Debug Tag Register Low is read using `mfspr` and is set by `dcread`.

![DCDBTRL](image)

Figure 88. Data Cache Debug Tag Register Low

### Programming Note
An example implementation of DCDBTRL could have the following content and format.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:44</td>
<td>Reserved (TRA)</td>
</tr>
<tr>
<td>45</td>
<td><strong>U bit parity</strong> (UPAR)</td>
</tr>
<tr>
<td>46:47</td>
<td><strong>Tag parity</strong> (TPAR)</td>
</tr>
<tr>
<td>48:51</td>
<td><strong>Data parity</strong> (DPAR)</td>
</tr>
<tr>
<td>52:55</td>
<td><strong>Modified</strong> (dirty) <strong>parity</strong> (MPAR)</td>
</tr>
<tr>
<td>56:59</td>
<td><strong>Dirty Indicators</strong> (D)</td>
</tr>
<tr>
<td></td>
<td>The “dirty” (modified) indicators for each of the four doublewords in the cache block</td>
</tr>
<tr>
<td>60</td>
<td><strong>U0 Storage Attribute</strong> (U0)</td>
</tr>
<tr>
<td></td>
<td>The U0 storage attribute for the page associated with this cache block</td>
</tr>
<tr>
<td>61</td>
<td><strong>U1 Storage Attribute</strong> (U1)</td>
</tr>
<tr>
<td></td>
<td>The U1 storage attribute for the page associated with this cache block</td>
</tr>
<tr>
<td>62</td>
<td><strong>U2 Storage Attribute</strong> (U2)</td>
</tr>
<tr>
<td></td>
<td>The U2 storage attribute for the page associated with this cache block</td>
</tr>
<tr>
<td>63</td>
<td><strong>U3 Storage Attribute</strong> (U3)</td>
</tr>
<tr>
<td></td>
<td>The U3 storage attribute for the page associated with this cache block</td>
</tr>
</tbody>
</table>

Implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.
### A.2.1.3 Instruction Cache Debug Data Register

The Instruction Cache Debug Data Register (ICDBDR) is a read-only 32-bit Special Purpose Register. The Instruction Cache Debug Data Register can be read using `mfspr` and is set by `icread`.

![ICDBDR](image)

This register is hypervisor privileged.

### A.2.1.4 Instruction Cache Debug Tag Register High

The Instruction Cache Debug Tag Register High (ICDBTRH) is a 32-bit Special Purpose Register. The Instruction Cache Debug Tag Register High is read using `mfspr` and is set by `icread`.

![ICDBTRH](image)

This register is hypervisor privileged.

### Programming Note

An example implementation of ICDBTRH could have the following content and format.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:55</td>
<td>Tag Effective Address (TEA)</td>
</tr>
<tr>
<td></td>
<td>Bits 0:23 of the 32-bit effective address associated with this cache block</td>
</tr>
<tr>
<td>56</td>
<td>Valid (V)</td>
</tr>
<tr>
<td></td>
<td>The valid indicator for the cache block (1 indicates valid)</td>
</tr>
<tr>
<td>57:58</td>
<td>Tag parity (TPAR)</td>
</tr>
<tr>
<td>59</td>
<td>Instruction Data parity (DPAR)</td>
</tr>
<tr>
<td>60:63</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.

### A.2.1.5 Instruction Cache Debug Tag Register Low

The Instruction Cache Debug Tag Register Low (ICDBTRL) is a 32-bit Special Purpose Register. The Instruction Cache Debug Tag Register Low is read using `mfspr` and is set by `icread`.

![ICDBTRL](image)

This register is hypervisor privileged.

### Programming Note

An example implementation of ICDBTRL could have the following content and format.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:53</td>
<td>Reserved</td>
</tr>
<tr>
<td>54</td>
<td>Translation Space (TS)</td>
</tr>
<tr>
<td></td>
<td>The address space portion of the virtual address associated with this cache block</td>
</tr>
<tr>
<td>55</td>
<td>Translation ID Disable (TD)</td>
</tr>
<tr>
<td></td>
<td>TID Disable field for the memory page associated with this cache block</td>
</tr>
<tr>
<td>56:63</td>
<td>Translation ID (TID)</td>
</tr>
<tr>
<td></td>
<td>TID field portion of the virtual address associated with this cache block</td>
</tr>
</tbody>
</table>

Other implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.
A.2.2 Embedded Cache Debug Instructions

**Data Cache Read X-form**

```plaintext
dcread RT,RA,RB
```

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
<th>RT</th>
<th>6</th>
<th>RA</th>
<th>11</th>
<th>RB</th>
<th>16</th>
<th>326</th>
<th>21</th>
<th>486</th>
</tr>
</thead>
</table>

[Alternative Encoding]

```plaintext
if RA = 0 then b ← 0
else
    b ← (RA)
EA ← b + (RB)
C ← log₂(cache size)
B ← log₂(cache block size)
IDX ← EA₆₄⁻₄₈⁻₃₂⁻₁₆⁻₈⁻₄⁻₂⁻₁⁻₀
WD ← EA₆₄⁻₄₈⁻₃₂⁻₂₄⁻₁₆⁻₈⁻₄⁻₂⁻₁⁻₀
RT₀⁻₃₁ ← undefined
RT₃₂⁻₆₃ ← (data cache data)[IDX]₉₋₆₃₋₆₁
DCDBTRH ← (data cache tag high)[IDX]
DCDBTRL ← (data cache tag low)[IDX]
```

Let the effective address (EA) be the sum of the contents of register RA, or 0 if RA is equal to 0, and the contents of register RB.

Let C = log₂(cache size in bytes).
Let B = log₂(cache block size in bytes).

EA₆₄⁻₄₈⁻₃₂⁻₁₆⁻₈⁻₄⁻₂⁻₁⁻₀ selects one of the 2ᶜ⁻ᵇ data cache blocks.

EA₆₄⁻₄₈⁻₃₂⁻₂₄⁻₁₆⁻₈⁻₄⁻₂⁻₁⁻₀ selects one of the data words in the selected data cache block.

The selected word in the selected data cache block is placed into register RT.

The contents of the data cache directory entry associated with the selected data cache block are placed into DCDBTRH and DCDBTRL (see Figure 87 and Figure 88).

*dcread* requires software to guarantee execution synchronization before subsequent *mfspr* instructions can read the results of the *dcread* instruction into GPRs. In order to guarantee that the *mfspr* instructions obtain the results of the *dcread* instruction, a sequence such as the following must be used:

```plaintext
sync # ensure that all previous cache operations have completed
dcread regT,regA,regB# read cache information;
isync # ensure dcread completes before attempting to read results
mfspr regD,dcdbtrh # move high portion of tag into GPR D
mfspr regE,dcdbtrl # move low portion of tag into GPR E
```

This instruction is hypervisor privileged.

**Special Registers Altered:**

```
DCDBTRH DCDBTRL
```

---

**Programming Note**

*dcread* can be used by a debug tool to determine the contents of the data cache, without knowing the specific addresses of the blocks which are currently contained within the cache.

---

**Programming Note**

Execution of *dcread* before the data cache has completed all cache operations associated with previously executed instructions (such as block fills and block flushes) is undefined.
**Instruction Cache Read X-form**

**icread** RA, RB

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>21</th>
<th>998</th>
</tr>
</thead>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + (RB)
C ← \(\log_2(\text{cache size})\)
B ← \(\log_2(\text{cache block size})\)
IDX ← \(EA_{64-C:63-B}\)
WD ← \(EA_{64-B:61}\)
ICDBDR ← \(\text{(instruction cache data)[IDX]}\)
ICDBTRH ← \(\text{(instruction cache tag high)[IDX]}\)
ICDBTRL ← \(\text{(instruction cache tag low)[IDX]}\)

Let the effective address (EA) be the sum of the contents of register RA, or 0 if RA is equal to 0, and the contents of register RB.

Let C = \(\log_2(\text{cache size in bytes})\).
Let B = \(\log_2(\text{cache block size in bytes})\).

\(EA_{64-C:63-B}\) selects one of the \(2^{C-B}\) instruction cache blocks.

\(EA_{64-B:61}\) selects one of the data words in the selected instruction cache block.

The selected word in the selected instruction cache block is placed into ICDBDR.

The contents of the instruction cache directory entry associated with the selected cache block are placed into ICDBTRH and ICDBTRL (see Figure 90 and Figure 91).

**icread** requires software to guarantee execution synchronization before subsequent **mfspr** instructions can read the results of the **icread** instruction into GPRs. In order to guarantee that the **mfspr** instructions obtain the results of the **icread** instruction, a sequence such as the following must be used:

```
icread   regA, regB  # read cache information
isync     # ensure icread completes
          # before attempting to
          # read results
mficdbdr  regC  # move instruction
          # information into GPR C
mficdbtrh regD  # move high portion of
          # tag into GPR D
mficdbtrl regE  # move low portion of tag
          # into GPR E
```

This instruction is hypervisor privileged.

**Special Registers Altered:**

ICDBDR  ICDBTRH  ICDBTRL

---

**Programming Note**

**icread** can be used by a debug tool to determine the contents of the instruction cache, without knowing the specific addresses of the blocks which are currently contained within the cache.
Appendix B. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book III.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.
B.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the \texttt{mtspr} and \texttt{mfspr} instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, and for the \textit{Move From Time Base} instruction defined in Book II.

The \texttt{mtspr} and \texttt{mfspr} instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the \textit{Move From Time Base} instruction, which specifies the portion of the Time Base as a numeric operand.

Note: \texttt{mftb} serves as both a basic and an extended mnemonic. The Assembler will recognize an \texttt{mftb} mnemonic with two operands as the basic form, and an \texttt{mftb} mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|l|}
\hline
\textbf{Special Purpose Register} & \textbf{Move To SPR} & \textbf{Move From SPR} \\
\hline
\multicolumn{5}{|c|}{\textbf{Extended}} \\
\hline
\textbf{Fixed-Point Exception Register} & \texttt{mtxer Rx} & \texttt{mtspr 1,Rx} & \texttt{mtxer Rx} & \texttt{mfspr Rx,1} \\
\textbf{Link Register} & \texttt{mtlr Rx} & \texttt{mtspr 8,Rx} & \texttt{mtlr Rx} & \texttt{mfspr Rx,8} \\
\textbf{Count Register} & \texttt{mtctr Rx} & \texttt{mtspr 9,Rx} & \texttt{mtctr Rx} & \texttt{mfspr Rx,9} \\
\textbf{Decrementer} & \texttt{mddec Rx} & \texttt{mtspr 22,Rx} & \texttt{mddec Rx} & \texttt{mfspr Rx,22} \\
\textbf{Save/Restore Register 0} & \texttt{mtsr0 Rx} & \texttt{mtspr 26,Rx} & \texttt{mtsr0 Rx} & \texttt{mfspr Rx,26} \\
\textbf{Save/Restore Register 1} & \texttt{mtsr1 Rx} & \texttt{mtspr 27,Rx} & \texttt{mtsr1 Rx} & \texttt{mfspr Rx,27} \\
\textbf{Special Purpose Registers G0 through G3} & \texttt{mtsprg n,Rx} & \texttt{mtspr 272+n,Rx} & \texttt{mtsprg Rx,n} & \texttt{mfspr Rx,272+n} \\
\textbf{Time Base [Lower]} & \texttt{mttbl Rx} & \texttt{mtspr 284,Rx} & \texttt{mftb Rx} & \texttt{mftb Rx,268$^1$} \\
\textbf{Time Base Upper} & \texttt{mtbpu Rx} & \texttt{mtspr 285,Rx} & \texttt{mftb Rx} & \texttt{mftb Rx,269$^1$} \\
\textbf{PPR32} & \texttt{mtprr32 Rx} & \texttt{mtspr 898, Rx} & \texttt{mftpr32 Rx} & \texttt{mfspr Rx,898} \\
\textbf{Processor Version Register} & - & - & \texttt{mpvpr Rx} & \texttt{mfspr Rx,287} \\
\hline
\end{tabular}
\caption{Extended mnemonics for moving to/from an SPR}
\end{table}

$^1$ The \texttt{mftb} instruction is Category: Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an \texttt{mfspr} instruction for the \texttt{mftb} and \texttt{mftbu} extended mnemonics; see the corresponding Assembler Note in the \texttt{mftb} instruction description (see Section 6.2.1 of Book II).
B.2 Data Cache Block Flush
Mnemonics [Category: Embedded.Phased In]

The L field in the Data Cache Block Flush by External PID instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

**Note:** `dcbfep` serves as both a basic and an extended mnemonic. The Assembler will recognize a `dcbfep` mnemonic with three operands as the basic form, and a `dcbflep` mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

- `dcbfep RA,RB` (equivalent to: `dcbfep RA,RB,0`)
- `dcbflep RA,RB` (equivalent to: `dcbfep RA,RB,1`)
- `dcblfpep RA,RB` (equivalent to: `dcbfep RA,RB,3`
Appendix C. Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations

C.1 Hardware Guidelines

C.1.1 64-bit Specific Instructions
The instructions in the Category: 64-Bit are considered restricted only to 64-bit processing. A 32-bit implementation need not implement the group; likewise, the 32-bit applications will not utilize any of these instructions. All other instructions shall either be supported directly by the implementation, or sufficient infrastructure will be provided to enable software emulation of the instructions. A 64-bit implementation that is executing in 32-bit mode may choose to take an Unimplemented Instruction Exception when these 64-bit specific instructions are executed.

C.1.2 Registers on 32-bit Implementations
The Power ISA provides 32-bit and 64-bit registers. All 32-bit registers shall be supported as defined in the specification except the MSR and EPCR. The MSR shall be supported as defined in the specification except that CM is treated as a reserved bit. EPCR shall be supported as defined in the specification except that ICM and GICM are treated as reserved bits. Only bits 32:63 of the 64-bit registers are required to be implemented in hardware in a 32-bit implementation except for the 64-bit FPRs. Such 64-bit registers include the LR, the CTR, the XER, the 32 GPRs, SRR0, CSRR0, DSRR0 <E.ED>, MCSRR0, and GSRR0 <E.HV>.

For additional information, see Section 1.5.2 of Book I.

C.1.3 Addressing on 32-bit Implementations
Only bits 32:63 of the 64-bit instruction and data storage effective addresses need to be calculated and presented to main storage. Given that the only branch and data storage access instructions that are not included in Section C.1.1 are defined to prepend 32 0s to bits 32:63 of the effective address computation, a 32-bit implementation can simply bypass the prepending of the 32 0s when implementing these instructions. For Branch to Link Register and Branch to Count Register instructions, given the LR and CTR are implemented only as 32-bit registers, only concatenating 2 0s to the right of bits 32:61 of these registers is necessary to form the 32-bit branch target address.

For next sequential instruction address computation, the behavior is the same as for 64-bit implementations in 32-bit mode.

C.1.4 TLB Fields on 32-bit Implementations
32-bit implementations should support bits 32:53 of the Effective Page Number (EPN) field in the TLB. This size provides support for a 32-bit effective address, which Power ISA ABIs may have come to expect to be available. 32-bit implementations may support greater than 32-bit real addresses by supporting more than bits 32:53 of the Real Page Number (RPN) field in the TLB.

C.1.5 Thread Control and Status on 32-bit Implementations
As the TEN and TENSOR are 32-bits on 32-bit implementations, the maximum number of threads for such implementations is limited to 32.

C.2 32-bit Software Guidelines

C.2.1 32-bit Instruction Selection
Any software that uses any of the instructions listed in Category: 64-Bit shall be considered 64-bit software, and correct execution cannot be guaranteed on 32-bit implementations. Generally speaking, 32-bit software should avoid using any instruction or instructions that depend on any particular setting of bits 0:31 of any 64-bit application-accessible system register, including General Purpose Registers, for producing the correct 32-bit results. Context switching may or may not preserve the upper 32 bits of application-accessible 64-bit
system registers and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.
Appendix D. Example Performance Monitor
[Category: Embedded.Performance Monitor]

D.1 Overview

This appendix describes an example of a Performance Monitor facility. It defines an architecture suitable for performance monitoring facilities in the Embedded environment. The architecture itself presents only programing model visible features in conjunction with architecturally defined behavioral features. Much of the selection of events is by necessity implementation-dependent and is not described as part of the architecture; however, this document provides guidelines for some features of a performance monitor implementation that should be followed by all implementations.

The example Performance Monitor facility provides the ability to monitor and count predefined events such as clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the Performance Monitor exception. While most of the specific events are not architected, the mechanism of controlling data collection is.

The example Performance Monitor facility can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize performance in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

D.2 Programming Model

The example Performance Monitor facility defines a set of Performance Monitor Registers (PMRs) that are used to collect and control performance data collection and an interrupt to allow intervention by software. The PMRs provide various controls and access to collected data. They are categorized as follows:

- Counter registers. These registers are used for data collection. The occurrence of selected events are counted here. These registers are named PMC0..15. User and supervisor level access to these registers is through different PMR numbers allowing different access rights.
- Global controls. This register control global settings of the Performance Monitor facility and affect all counters. This register is named PMGC0. User and supervisor level access to these registers is through different PMR numbers allowing different access rights. In addition, a bit in the MSR (MSRPM) is defined to enable/disable counting.
- Local controls. These registers control settings that apply only to a particular counter. These registers are named PMLCa0..15 and PMLCb0..15. User and supervisor level access to these registers is through different PMR numbers allowing different access rights. Each set of local control registers (PMLCa n and PMLCb n) contains controls that apply to the associated same numbered counter register (e.g. PMLCa0 and PMLCb0 contain controls for PMC0 while PMLCa1 and PMLCb1 contain controls for PMC1).

**Assembler Note**

The counter registers, global controls, and local controls have alias names which cause the assembler to use different PMR numbers. The names PMC0...15, PMGC0, PMLCa0...15, and PMLCb0...15 cause the assembler to use the supervisor level PMR number, and the names UPMC0...15, UPMGC0, UPMLCa0...15, and UPMLCb0...15 cause the assembler to use the user-level PMR number.

**Architecture Note**

The two mark values (0 and 1) are equivalent except with respect to interrupts. That is, either mark value can be specified for a given process, and either mark value can control whether the PMCs are incremented, but interrupts always cause the mark value in the MSR to be set to 0.
A given implementation may implement fewer counter registers (and their associated control registers) than are architected. Architected counter and counter control registers that are not implemented behave the same as unarchitected Performance Monitor Registers. PMRs are described in Section D.3.

Software uses the global and local controls to select which events are counted in the counter registers, when such events should be counted, and what action should be taken when a counter overflows. Software can use the collected information to determine performance attributes of a given segment of code, a process, or the entire software system. PMRs can be read by software using the `mfpmr` instruction and PMRs can be written by using the `mtpmr` instruction. Both instructions are described in Section D.4.

Since counters are defined as 32-bit registers, it is possible for the counting of some events to overflow. A Performance Monitor interrupt is provided that can be programmed to occur in the event of a counter overflow. The Performance Monitor interrupt is described in detail in Section D.2.5 and Section D.2.6.

### D.2.1 Event Counting

Event counting can be configured in several different ways. This section describes configurability and specific unconditional counting modes.

### D.2.2 Thread Context Configurability

Counting can be enabled if conditions in the thread state match a software-specified condition. Because a software task scheduler may switch a thread's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The Performance Monitor mark bit, MSR_PMM, is used for this purpose. System software may set this bit to 1 when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR_PR and MSR_PMM together define a state that the thread (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by the PMLCan_FCS, PMLCan_FCU, PMLCan_FCM1, and PMLCan_FCM0 fields in PMLCan (the state for which monitoring is enabled), counting is enabled for PMCn.

Each event, on an implementation basis, may count regardless of the value of MSR_PMM. The counting behavior of each event should be documented in the User's Manual.

The thread states and the settings of the PMLCan_FCS, PMLCan_FCU, PMLCan_FCM1 and PMLCan_FCM0 fields in PMLCan necessary to enable monitoring of each thread state are shown in Figure 92.

<table>
<thead>
<tr>
<th>Thread State</th>
<th>FCS</th>
<th>FCU</th>
<th>FCM1</th>
<th>FCM0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marked</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Not marked</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Supervisor</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>User</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Marked and supervisor</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Marked and user</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Not marked and supervisor</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Not mark and user</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>All</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>None</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>None</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 92. Thread States and PMLCan Bit Settings

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR_PMM and MSR_PR. This can be accomplished by setting PMLCan_FCS, PMLCan_FCU, PMLCan_FCM1, and PMLCan_FCM0 to 0 for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR_PMM and MSR_PR. This can be accomplished by setting PMGC0FAC to 1 or by setting PMLCan_FCU to 1 for each counter control. Alternatively, this can be accomplished by setting PMLCan_FCM1 to 1 and PMLCan_FCM0 to 1 for each counter control or by setting PMLCan_FCS to 1 and PMLCan_FCU to 1 for each counter control.

---

**Programming Note**

Events may be counted in a fuzzy manner. That is, events may not be counted precisely due to the nature of an implementation. Users of the Performance Monitor facility should be aware that an event may be counted even if it was precisely filtered, though it should not have been. In general such discrepancies are statistically unimportant and users should not assume that counts are explicitly accurate.

### D.2.3 Event Selection

Events to count are determined by placing an implementation defined event value into the PMLCan0..15EVENT field. Which events may be programmed into which counter are implementation specific and should be defined in the User’s Manual. In general, most events may be programmed into any of the implementation available counters. Programming a counter with an event that is not supported for that counter gives boundedly undefined results.
D.2.4 Thresholds

Thresholds are values that must be exceeded for an event to be counted. Threshold values are programmed in the PMLCb0..15.THRESHOLD field. The events which may be thresholded and the units of each event that may be thresholded are implementation-dependent. Programming a threshold value for an event that is not defined to use a threshold gives boundedly undefined results.

D.2.5 Performance Monitor Exception

A Performance Monitor exception occurs when counter overflow detection is enabled and a counter overflows. More specifically, for each counter register \(n\), if \(\text{PMGC0\_RMIE}=1\), \(\text{PMLCn\_CE}=1\), \(\text{PMCR\_OV}=1\), and the performance monitor interrupt is enabled in the MSR (see below), a Performance Monitor exception is said to exist. The Performance Monitor exception condition will cause a Performance Monitor interrupt if the exception is the highest priority exception.

The performance monitor interrupt enabling conditions in the MSR are as follows:

- If category E.HV is implemented, the performance monitor interrupt is enabled if:
  - \(\text{EPCRPMGS}=0\) and \(\text{MSR\_EE}=1\) or \(\text{MSR\_GS}=1\). The interrupt will be directed to the hypervisor state. or if
  - \(\text{EPCRPMGS}=1\) and \(\text{MSR\_EE}=1\) and \(\text{MSR\_GS}=1\). The interrupt will be directed to the guest state.
- If category E.HV is not implemented, the performance monitor interrupt is enabled if \(\text{MSR\_EE}=1\).

The Performance Monitor exception is level sensitive and the exception condition may cease to exist if any of the required conditions fail to be met. Thus it is possible for a counter to overflow and continue counting events until \(\text{PMCR\_OV}\) becomes 0 without taking a Performance Monitor interrupt if the enabling conditions in the MSR are not met during the overflow condition. To avoid this, software should program the counters to freeze if an overflow condition is detected (see Section D.3.4).

D.2.6 Performance Monitor Interrupt

A Performance Monitor interrupt occurs when a Performance Monitor exception exists and no higher priority exception exists. When a Performance Monitor interrupt occurs, SRR0 and SRR1 (GSR0 and GS1 if \(\text{EPCRPMGS}=1\ <\text{E.HV}\)) record the current state of the NIA and the MSR and the MSR is set to handle the interrupt. Instruction execution resumes at an address based on which categories are supported:

- If both Interrupt Fixed Offsets [Category: Embedded.Phased-In] and Embedded.Hypervisor are supported, instruction execution resumes at:
  - address IVPR0:51 || 0x260 when \(\text{EPCRPMGS}=0\). Otherwise, instruction execution resumes at address GIVPR0:47 || 0x260.
  - If Interrupt Fixed Offsets [Category: Embedded.Phased-In] is supported and Embedded.Hypervisor is not supported, instruction execution resumes at address IVPR0:51 || 0x260.
  - If Interrupt Fixed Offsets [Category: Embedded.Phased-In] is not supported and Embedded.Hypervisor is supported, instruction execution resumes at:
    - address IVPR0:51 || IVOR3548:59 || 0b0000. when \(\text{EPCRPMGS}=0\). Otherwise, instruction execution resumes at address GIVPR0:47 || GIVOR3548:59 || 0b0000.
  - If neither Interrupt Fixed Offsets [Category: Embedded.Phased-In] nor Embedded.Hypervisor is supported, instruction execution resumes at address IVPR0:51 || IVOR3548:59 || 0b0000.

The Performance Monitor interrupt is precise and asynchronous.

---

**Programming Note**

When taking a Performance Monitor interrupt software should clear the overflow condition by reading the counter register and setting the counter register to a non-overflow value since the normal return from the interrupt will set MSR\_EE back to 1.

D.3 Performance Monitor Registers

D.3.1 Performance Monitor Global Control Register 0

The Performance Monitor Global Control Register 0 (PMGC0) controls all Performance Monitor counters.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMGC0</td>
<td>32-63</td>
</tr>
</tbody>
</table>

Figure 93. [User] Performance Monitor Global Control Register 0

These bits are interpreted as follows:
Freeze All Counters (FAC)
The FAC bit is sticky; that is, once set to 1 it remains set to 1 until it is set to 0 by an `mtpr` instruction.
- 0 The PMCs can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMCs cannot be incremented.

Performance Monitor Interrupt Enable (PMIE)
- 0 Performance Monitor interrupts are disabled.
- 1 Performance Monitor interrupts are enabled and occur when an enabled condition or event occurs. Enabled conditions and events are described in Section D.2.5.

Freeze Counters on Enabled Condition or Event (FCECE)
Enabled conditions and events are described in Section D.2.5.
- 0 The PMCs can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMCs can be incremented (if enabled by other Performance Monitor control fields) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0FAC is set to 1. It is the user’s responsibility to set PMGC0FAC to 0.

Freeze Counter in Supervisor State (FCS)
- 0 The PMC is incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC cannot be incremented if MSRPR is 0.

Freeze Counter in User State (FCU)
- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC cannot be incremented if MSRPR is 1.

Freeze Counter while Mark is Set (FCM1)
- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC cannot be incremented if MSRPMM is 1.

Freeze Counter while Mark is Cleared (FCM0)
- 0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC cannot be incremented if MSRPMM is 0.

Condition Enable (CE)
- 0 Overflow conditions for PMC cannot occur (PMC cannot cause interrupts, cannot freeze counters)
- 1 Overflow conditions occur when the most-significant-bit of PMC is equal to 1.

It is recommended that CE be set to 0 when counter PMC is selected for chaining; see Section D.5.1.

Freeze Counter in Guest State (FCGS)
- 0 The PMC is incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC cannot be incremented if MSRGS is 1.

Freeze Counter in Hypervisor State (FCHS)
- 0 The PMC is incremented (if enabled by other Performance Monitor control fields).
- 1 The PMC cannot be incremented if MSRGS is 0 and MSRPR is 0.

Event Selector (EVENT)
Up to 128 events selectable; see Section D.2.3.

Setting is implementation-dependent.
The UPMLCa0..15 registers are aliases to the PMLCa0..15 registers for user mode read only access.

D.3.3 Performance Monitor Local Control B Registers

The Performance Monitor Local Control B Registers 0 through 15 (PMLCb0..15) specify a threshold value and a multiple to apply to a threshold event selected for the corresponding Performance Monitor counter. Threshold capability is implementation counter dependent. Not all events or all counters of an implementation are guaranteed to support thresholds. PMLCb works with the corresponding numbered PMLCa register.

![Figure 95. Performance Monitor Local Control B Register](image)

PMLCb is set to 0 at reset. These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>52:32</td>
<td>Reserved</td>
</tr>
<tr>
<td>51:53</td>
<td>Threshold Multiple (THRESHMUL)</td>
</tr>
<tr>
<td></td>
<td>000: Threshold field is multiplied by 1 (THRESHOLD × 1)</td>
</tr>
<tr>
<td></td>
<td>001: Threshold field is multiplied by 2 (THRESHOLD × 2)</td>
</tr>
<tr>
<td></td>
<td>010: Threshold field is multiplied by 4 (THRESHOLD × 4)</td>
</tr>
<tr>
<td></td>
<td>011: Threshold field is multiplied by 8 (THRESHOLD × 8)</td>
</tr>
<tr>
<td></td>
<td>100: Threshold field is multiplied by 16 (THRESHOLD × 16)</td>
</tr>
<tr>
<td></td>
<td>101: Threshold field is multiplied by 32 (THRESHOLD × 32)</td>
</tr>
<tr>
<td></td>
<td>110: Threshold field is multiplied by 64 (THRESHOLD × 64)</td>
</tr>
<tr>
<td></td>
<td>111: Threshold field is multiplied by 128 (THRESHOLD × 128)</td>
</tr>
<tr>
<td>62:56</td>
<td>Reserved</td>
</tr>
<tr>
<td>61:63</td>
<td>Threshold (THRESHOLD)</td>
</tr>
</tbody>
</table>

By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can measure the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

![Programming Note](image)

The UPMLCb0..15 registers are aliases to the PMLCb0..15 registers for user mode read only access.

D.3.4 Performance Monitor Counter Registers

The Performance Monitor Counter Registers (PMC0..15) are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count up to 128 events.

![Figure 96. Performance Monitor Counter Registers](image)

PMCs are set to 0 at reset. These bits are interpreted as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Overflow (OV)</td>
</tr>
<tr>
<td></td>
<td>0 Counter has not reached an overflow state.</td>
</tr>
<tr>
<td></td>
<td>1 Counter has reached an overflow state.</td>
</tr>
<tr>
<td>62:33</td>
<td>Counter Value (CV)</td>
</tr>
<tr>
<td></td>
<td>Indicates the number of occurrences of the specified event.</td>
</tr>
</tbody>
</table>

The minimum value for a counter is 0 (0x0000_0000) and the maximum value is 4,294,967,295 (0xFFFF_FFFF). A counter can increment up to the maximum value and then wraps to the minimum value. A counter enters the overflow state when the high-order bit is set to 1, which normally occurs only when the counter increments from a value below 2,147,483,648 (0x8000_0000) to a value greater than or equal to 2,147,483,648 (0x8000_0000).

Several different actions may occur when an overflow state is reached, depending on the configuration:

- If PMLCa\_CE is 0, no special actions occur on overflow: the counter continues incrementing, and no exception is signaled.
- If PMLCa\_CE and PMG0\_FCECE are 1, all counters are frozen when PMCn overflows.
- If PMLCa\_CE, PMG0\_PMIE, and MSR\_EE are 1, an exception is signaled when PMCn reaches over-
flow. Note that the interrupts are masked by setting MSR_{EE} to 0. An overflow condition may be present while MSR_{EE} is zero, but the interrupt is not taken until MSR_{EE} is set to 1.

If an overflow condition occurs while MSR_{EE} is 0 (the exception is masked), the exception is still signalled once MSR_{EE} is set to 1 if the overflow condition is still present and the configuration has not been changed in the meantime to disable the exception; however, if MSR_{EE} remains 0 until after the counter leaves the overflow state (MSB becomes 0), or if MSR_{EE} remains 0 until after PMLCn_{CE} or PMGC0_{PMIE} are set to 0, the exception does not occur.

**Programming Note**

Loading a PMC with an overflowed value can cause an immediate exception. For example, if PMLCn_{CE}, PMGC0_{PMIE}, and MSR_{EE} are all 1, and an *mtpmr* loads an overflowed value into a PMCn that previously held a non-overflowed value, then an interrupt will be generated before any event counting has occurred.

The following sequence is generally recommended for setting the counter values and configurations.

1. Set PMGC0_{FAC} to 1 to freeze the counters.
2. Perform a series of *mtpmr* operations to initialize counter values and configure the control registers
3. Release the counters by setting PMGC0_{FAC} to 0 with a final *mtpmr*. 
D.4 Performance Monitor Instructions

**Move From Performance Monitor Register**

XFX-form

```
movefrom pmr

rt,pmr
```

**Move To Performance Monitor Register**

XFX-form

```
moveto pmr

rt,pmr
```

n ← pmrn5:9 || pmrn0:4
if length(PMR(n)) = 64 then
 RT ← PMR(n)
else
 RT ← 32:63 || PMR(n)32:63

Let PMRN denote a Performance Monitor Register number and PMR the set of Performance Monitor Registers.

The contents of the designated Performance Monitor Register are placed into register RT.

The list of defined Performance Monitor Registers and their privilege class is provided in Figure 97.

Execution of this instruction specifying a defined and privileged Performance Monitor Register when MSRPR=1 will result in a Privileged Instruction exception.

Category: Embedded.Hypervisor

If MSRPM = 1 and MSRGS = 1, execution of this instruction specifying a defined Performance Monitor Register sets RT to 0.

Execution of this instruction specifying an undefined Performance Monitor Register will either result in an Illegal Instruction exception or will produce an undefined value for register RT.

Special Registers Altered:

None

<table>
<thead>
<tr>
<th>decimal</th>
<th>PMR</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Cat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pmrn5:9</td>
<td>pmrn0:4</td>
<td>mtpmr</td>
<td>mtfpmr</td>
</tr>
<tr>
<td>0-15</td>
<td>00000</td>
<td>0xxxx</td>
<td>PMC0..15</td>
<td>-</td>
</tr>
<tr>
<td>16-31</td>
<td>00000</td>
<td>1xxxx</td>
<td>PMC0..15</td>
<td>yes</td>
</tr>
<tr>
<td>128-143</td>
<td>00100</td>
<td>0xxxx</td>
<td>PMLCA0..15</td>
<td>-</td>
</tr>
<tr>
<td>144-159</td>
<td>00100</td>
<td>1xxxx</td>
<td>PMLCA0..15</td>
<td>yes</td>
</tr>
<tr>
<td>256-271</td>
<td>01000</td>
<td>0xxxx</td>
<td>PMLCB0..15</td>
<td>-</td>
</tr>
<tr>
<td>272-287</td>
<td>01000</td>
<td>1xxxx</td>
<td>PMLCB0..15</td>
<td>yes</td>
</tr>
<tr>
<td>384</td>
<td>01100</td>
<td>00000</td>
<td>PMGC0</td>
<td>-</td>
</tr>
<tr>
<td>400</td>
<td>01100</td>
<td>10000</td>
<td>PMGC0</td>
<td>yes</td>
</tr>
</tbody>
</table>

- This register is not defined for this instruction.

Figure 97. Embedded.Performance Monitor PMRs
D.5 Performance Monitor Software Usage Notes

D.5.1 Chaining Counters

An implementation may contain events that are used to "chain" counters together to provide a larger range of event counts. This is accomplished by programming the desired event into one counter and programming another counter with an event that occurs when the first counter transitions from 1 to 0 in the most significant bit.

The counter chaining feature can be used to decrease the processing pollution caused by Performance Monitor interrupts, (things like cache contamination, and pipeline effects), by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's carry-out event acts like a carry-out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:
    mfpmr Rx,pmctr1 #load from upper counter
    mfpmr Ry,pmctr0 #load from lower counter
    mfpmr Rz,pmctr1 #load from upper counter
    cmp cr0,0,Rz,Rx #see if 'old' = 'new'
    bc 4,2,loop
        #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

D.5.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. Assume an example event, dLFB load miss cycles, requires a threshold value. A dLFB load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. If the event is counted on two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.
Book VLE:

Power ISA Operating Environment Architecture -
Variable Length Encoding (VLE) Environment
[Category: Variable Length Encoding]
Chapter 1. Variable Length Encoding Introduction

This chapter describes computation modes, document conventions, a processor overview, instruction formats, storage addressing, and instruction addressing.

1.1 Overview

Variable Length Encoding (VLE) is a code density optimized re-encoding of much of the instruction set defined by Books I, II, and III-E using both 16-bit and 32-bit instruction formats.

VLE offers more efficient binary representations of applications for the Embedded processor spaces where code density plays a major role in affecting overall system cost, and to a somewhat lesser extent, performance.

VLE is a supplement to the instruction set defined by Book I-III and code pages using VLE encoding or non-VLE encoding can be intermingled in a system providing focus on both high performance and code density where most needed.

VLE provides alternative encodings to instructions defined in Books I-III to enable reduced code footprint. This set of alternative encodings is selected on a page basis. A single storage attribute bit selects between standard instruction encodings and VLE instructions for that page of memory.

Instruction encodings in pages marked as VLE are either 16 or 32 bits long, and are aligned on 16-bit boundaries. Because of this, all instruction pages marked as VLE are required to use Big-Endian byte ordering.

The programming model uses the same register set with both instruction set encodings, although some registers are not accessible by VLE instructions using the 16-bit formats and not all condition register (CR) fields are used by Conditional Branch instructions or instructions that access the condition register executing from a VLE instruction page. In addition, immediate fields and displacements differ in size and use, due to the more restrictive encodings imposed by VLE instruction formats.

VLE additional instruction fields are described in Section 1.4.19, “Instruction Fields”.

Other than the requirement of Big-Endian byte ordering for instruction pages and the additional storage attribute to identify whether the instruction page corresponds to a VLE section of code, VLE complies with the memory model, register model, timer facilities, debug facilities, and interrupt/exception model defined in Book I-III and therefore execute in the same environment as non-VLE instructions.

1.2 Documentation Conventions

Book VLE adheres to the documentation conventions defined in Section 1.3 of Book I. Note however that this book defines instructions that apply to the User Instruction Set Architecture, the Virtual Environment Architecture, and the Operating Environment Architecture.

1.2.1 Description of Instruction Operation

The RTL (register transfer language) descriptions in Book VLE conform to the conventions described in Section 1.3.4 of Book I.

1.3 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. VLE instruction semantics are either identical or similar to those of other instructions in the architecture. Where the semantics, side-effects, and binary encodings are identical, the standard mnemonics and formats are used. Such unchanged instructions are listed and appropriately referenced, but the instruction definitions are not replicated in this book. Where the semantics are similar but the binary encodings differ, the standard mnemonic is typically preceded with an _e_ to denote a VLE instruction. To distinguish between similar instructions available in both 16- and 32-bit forms under VLE and standard instructions, VLE instructions encoded with 16 bits have an _se_ prefix. The following are examples:
1.4 VLE Instruction Formats

All VLE instructions to be executed are either two or four bytes long and are halfword-aligned in storage. Thus, whenever instruction addresses are presented to the processor (as in Branch instructions), the low-order bit is treated as 0. Similarly, whenever the processor generates an instruction address, the low-order bit is zero.

The format diagrams given below show horizontally all valid combinations of instruction fields. Only those formats that are unique to VLE-defined instructions are included here. Instruction forms that are available in VLE or non-VLE mode are described in Section 1.6 of Book I and are not repeated here.

In some cases an instruction field must contain a particular value. If a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described for invalid instruction forms in Book I.

VLE instructions use split field notation as defined in Section 1.6 of Book I.

1.4.4 OIM5-form (16-bit register + offset immediate Instructions)

1.4.5 IM7-form (16-bit Load immediate Instructions)

1.4.6 R-form (16-bit Monadic Instructions)

1.4.7 RR-form (16-bit Dyadic Instructions)

1.4.8 SD4-form (16-bit Load/Store Instructions)

1.4.9 BD15-form
1.4.10 BD24-form

Figure 10. BD24 instruction format

1.4.11 D8-form

Figure 11. D8 instruction format

1.4.12 ESC-form

Figure 12. I16A instruction format

1.4.13 I16A-form

Figure 13. I16L instruction format

1.4.14 I16L-form

Figure 14. M instruction format

1.4.15 M-form

1.4.16 SCI8-form

Figure 15. SCI8 instruction format

1.4.17 LI20-form

Figure 16. LI20 instruction format

1.4.18 X-form

Figure 17. X instruction format

1.4.19 Instruction Fields

VLE uses instruction fields defined in Section 1.6.28 of Book I as well as VLE-defined instruction fields defined below.

ARX (12:15)
Field used to specify an “alternate” General Purpose Register in the range R8:R23 to be used as a destination.

ARY (8:11)
Field used to specify an “alternate” General Purpose Register in the range R8:R23 to be used as a source.

BD8 (8:15), BD15 (16:30), BD24 (7:30)
Immediate field specifying a signed two's complement branch displacement which is concatenated on the right with 0b0 and sign-extended to 64 bits.

BD15. (Used by 32-bit branch conditional class instructions) A 15-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BD24. (Used by 32-bit branch class instructions) A 24-bit signed displacement that is
sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BD8. (Used by 16-bit branch and branch conditional class instructions) An 8-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BI16 (6:7), BI32 (12:15)
Field used to specify one of the Condition Register fields to be used as a condition of a Branch Conditional instruction.

BO16 (5), BO32 (10:11)
Field used to specify whether to branch if the condition is true, false, or to decrement the Count Register and branch if the Count Register is not zero in a Branch Conditional instruction.

BF32 (9:10)
Field used to specify one of the Condition Register fields to be used as a target of a compare instruction.

D8 (24:31)
The D8 field is a 8-bit signed displacement which is sign-extended to 64 bits.

ELEV (16:20)
Field used by the e_sc instruction.

F (21) Fill value used to fill the remaining 56 bits of a scaled-immediate 8 value.

LI20 (17:20 || 11:15 || 21:31)
A 20-bit signed immediate value which is sign-extended to 64 bits for the e_li instruction.

LK (7, 15, 31)
LINK bit.
0 Do not set the Link Register.
1 Set the Link Register. The sum of the value 2 or 4 and the address of the Branch instruction is placed into the Link Register.

OIM5 (7:11)
Offset Immediate field used to specify a 5-bit unsigned fixed-point value in the range [1:32] encoded as [0:31]. Thus the binary encoding of 0b00000 represents an immediate value of 1, 0b00001 represents an immediate value of 2, and so on.

OPCD (0:3, 0:4, 0:5, 0:9, 0:14, 0:15)
Primary opcode field.

Rc (6, 7, 20, 31)
RECORD bit.
0 Do not alter the Condition Register.
1 Set Condition Register Field 0.

RX (12:15)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

RY (8:11)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination for load/store data. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

RZ (8:11)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination for a load/store operation. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

SCL (22:23)
Field used to specify a scale amount in Immediate instructions using the SCI8-form. Scaling involves left shifting by 0, 8, 16, or 24 bits.

SD4 (4:7)
Used by 16-bit load and store class instructions. The SD4 field is a 4-bit unsigned immediate value zero-extended to 64 bits, shifted left according to the size of the operation, and then added to the base register to form a 64-bit EA. For byte operations, no shift is performed. For half-word operations, the immediate is shifted left one bit (concatenated with 0b0). For word operations, the immediate is shifted left two bits (concatenated with 0b0).SI (6:10 || 21:31, 11:15 || 21:31)
A 16-bit signed immediate value sign-extended to 64 bits and used as one operand of the instruction.

A 16-bit unsigned immediate value zero-extended to 64 bits or padded with 16 zeros and used as one operand of the instruction. The instruction encoding differs between the I16A and I16L instruction formats as shown in Section 1.4.13 and Section 1.4.14.

UI5 (7:11)
Immediate field used to specify a 5-bit unsigned fixed-point value.

UI7 (5:11)
Immediate field used to specify a 7-bit unsigned fixed-point value.

UI8 (24:31)
Immediate field used to specify an 8-bit unsigned fixed-point value.
For scaled immediate instructions using the \texttt{SCI8}-form, the instruction assembly syntax requires a single immediate value, \texttt{sci8}, that the assembler will synthesize into the appropriate \texttt{F}, \texttt{SCL}, and \texttt{UI8} fields. The \texttt{F}, \texttt{SCL}, and \texttt{UI8} fields must be able to be formed correctly from the given \texttt{sci8} value or the assembler will flag the assembly instruction as an error.
Chapter 2. VLE Storage Addressing

A program references memory using the effective address (EA) computed by the processor when it executes a Storage Access or Branch instruction (or certain other instructions described in Book II and Book III-E), or when it fetches the next sequential instruction.

2.1 Data Storage Addressing Modes

Table 1 lists data storage addressing modes supported by the VLE category.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base+16-bit displacement</td>
<td>D-form</td>
<td>The 16-bit D field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0 to produce the EA.</td>
</tr>
<tr>
<td>(32-bit instruction format)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base+8-bit displacement</td>
<td>D8-form</td>
<td>The 8-bit D8 field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0 to produce the EA.</td>
</tr>
<tr>
<td>(32-bit instruction format)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base+scaled 4-bit displacement</td>
<td>SD4-form</td>
<td>The 4-bit SD4 field zero-extended, scaled (shifted left) according to the size of the operand, and added to the contents of the GPR designated by RX to produce the EA. (Note that RX = 0 is not a special case.)</td>
</tr>
<tr>
<td>(16-bit instruction format)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base+Index</td>
<td>X-form</td>
<td>The GPR contents designated by RB are added to the GPR contents designated by RA or to zero if RA = 0 to produce the EA.</td>
</tr>
<tr>
<td>(32-bit instruction format)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2 Instruction Storage Addressing Modes

Table 2 lists instruction storage addressing modes supported by the VLE category.

<table>
<thead>
<tr>
<th>Mode Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taken BD24-form Branch instructions (32-bit instruction format)</td>
<td>The 24-bit BD24 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction.</td>
</tr>
<tr>
<td>Taken B15-form Branch instructions (32-bit instruction format)</td>
<td>The 15-bit BD15 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction to form the EA of the next instruction.</td>
</tr>
<tr>
<td>Taken BD8-form Branch instructions (16-bit instruction format)</td>
<td>The 8-bit BD8 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction to form the EA of the next instruction.</td>
</tr>
<tr>
<td>Sequential instruction fetching (or non-taken branch instructions)</td>
<td>The value 4 [2] is added to the address of the current 32-bit [16-bit] instruction to form the EA of the next instruction. If the address of the current instruction is 0xFFF_FFFF_FFFF_FFFC [0xFFF_FFFF_FFFF_FFFE] in 64-bit mode or 0xFFF_FFFF_FFFE in 32-bit mode, the address of the next sequential instruction is undefined.</td>
</tr>
<tr>
<td>Any Branch instruction with LK = 1 (32-bit instruction format)</td>
<td>The value 4 is added to the address of the current branch instruction and the result is placed into the LR. If the address of the current instruction is 0xFFF_FFFF_FFFF_FFFC in 64-bit mode or 0xFFF_FFFE in 32-bit mode, the result placed into the LR is undefined.</td>
</tr>
<tr>
<td>Branch se_bl, se_birl, se_bctrl instructions (16-bit instruction format)</td>
<td>The value 2 is added to the address of the current branch instruction and the result is placed into the LR. If the address of the current instruction is 0xFFF_FFFF_FFFF_FFFE in 64-bit mode or 0xFFF_FFFE in 32-bit mode, the result placed into the LR is undefined.</td>
</tr>
</tbody>
</table>

2.2.1 Misaligned, Mismatched, and Byte Ordering Instruction Storage Exceptions

A Misaligned Instruction Storage Exception occurs when an implementation which supports VLE attempts to execute an instruction that is not 32-bit aligned and the VLE storage attribute is not set for the page that corresponds to the effective address of the instruction. The attempted execution can be the result of a Branch instruction which has bit 62 of the target address set to 1 or the result of an rfi, se_rfi, rfcf, se_rfcf, rfdi, se_rfdi, rfgi, se_rfgi, rfmc, or se_rfmc instruction which has bit 62 set in SRR0, SRR0, CSRRO, CSRRO, DSRR0, DSRR0, GSRRO, GSRRO, MCSRR0, or MCSRRO respectively. If a Misaligned Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0(GSRR0) to the misaligned address for which execution was attempted.

A Byte Ordering Instruction Storage Exception occurs when an implementation which supports VLE attempts to execute an instruction that has the VLE storage attribute set to 1 and the E (Endian) storage attribute set to 1 for the page that corresponds to the effective address of the instruction. If a Byte Ordering Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0(GSRR0) to the address for which execution was attempted.

2.2.2 VLE Exception Syndrome Bits

Two bits in the Exception Syndrome Register (ESR) (see Section 7.2.13 of Book III-E) are provided to facilitate VLE exception handling, VLEMI and MIF.

ESR(GESR)_{VLEMI} is set when an exception and subsequent interrupt is caused by the execution or attempted
execution of an instruction that resides in memory with the VLE storage attribute set.

ESR(GESR)_MIF is set when an Instruction Storage Interrupt is caused by a Misaligned Instruction Storage Exception or when an Instruction TLB Error Interrupt was caused by a TLB miss on the second half of a misaligned 32-bit instruction.

ESR(GESR)_BO is set when an Instruction Storage Interrupt is caused by a Mismatched Instruction Storage Exception or a Byte Ordering Instruction Storage Exception.

### Programming Note

When an Instruction TLB Error Interrupt occurs as the result of a Instruction TLB miss on the second half of a 32-bit VLE instruction that is aligned to only 16-bits, SRR0 will point to the first half of the instruction and ESR_MIF will be set to 1. Any other status posted as a result of the TLB miss (such as MAS register updates described in Chapter 6 of Book III-E) will reflect the page corresponding to the second half of the instruction which caused the Instruction TLB miss.
Chapter 3. VLE Compatibility with Books I–III

This chapter addresses the relationship between VLE and Books I–III.

3.1 Overview

Category VLE uses the same semantics as Books I–III. Due to the limited instruction encoding formats, VLE instructions typically support reduced immediate fields and displacements, and not all operations defined by Books I–III are encoded in category VLE. The basic philosophy is to capture all useful operations, with most frequent operations given priority. Immediate fields and displacements are provided to cover the majority of ranges encountered in Embedded control code. Instructions are encoded in either a 16- or 32-bit format, and these may be freely intermixed.

VLE instructions cannot access floating-point registers (FPRs). VLE instructions use GPRs and SPRs with the following limitations:

- VLE instructions using the 16-bit formats are limited to addressing GPR0–GPR7, and GPR24–GPR31 in most instructions. Move instructions are provided to transfer register contents between these registers and GPR8–GPR23.
- VLE compare and bit test instructions using the 16-bit formats implicitly set their results in CR0.

VLE instruction encodings are generally different than instructions defined by Books I–III, except that most instructions falling within primary opcode 31 are encoded identically and have identical semantics unless they affect or access a resource not supported by category VLE.

3.2 VLE Processor and Storage Control Extensions

This section describes additional functionality to support category VLE.

3.2.1 Instruction Extensions

This section describes extensions to support VLE operations. Because instructions may reside on a half-word boundary, bit 62 is not masked by instructions that read an instruction address from a register, such as the LR, CTR, or a save/restore register 0, that holds an instruction address:

The instruction set defined by Books I–III is modified to support halfword instruction addressing, as follows:

- For Return From Interrupt instructions, such as rfi, rfci, rdfi, rfgi, and rfmc, no longer mask bit 62 of the respective save/restore register 0. The destination address is SRR0:62 \(\|\) 0b0, CSRR0:62 \(\|\) 0b0, DSRR0:62 \(\|\) 0b0, GSRR0:62 \(\|\) 0b0, and MCSRR0:62 \(\|\) 0b0, respectively.
- For bclr, bclrl, bcctr, and bcctl no longer mask bit 62 of the LR or CTR. The destination address is LR0:62 \(\|\) 0b0 or CTR0:62 \(\|\) 0b0.

3.2.2 MMU Extensions

VLE operation is indicated by the VLE storage attribute. When the VLE storage attribute for a page is set to 1, instruction fetches from that page are decoded and processed as VLE instructions. See Section 6.8.3 of Book III-E.

When instructions are executing from a page that has the VLE storage attribute set to 1, the processor is said to be in VLE mode.

3.3 VLE Limitations

VLE instruction fetches are valid only when performed in a Big-Endian mode. Attempting to fetch an instruction in a Little-Endian mode from a page with the VLE storage attribute set causes an Instruction Storage Byte-ordering exception.

Support for concurrent modification and execution of VLE instructions is implementation-dependent.
Chapter 4. Branch Operation Instructions

This section defines Branch instructions that can be executed when a processor is in VLE mode and the registers that support them.

4.1 Branch Facility Registers

The registers that support branch operations are:
- Section 4.1.1, “Condition Register (CR)”
- Section 4.1.2, “Link Register (LR)”
- Section 4.1.3, “Count Register (CTR)”

4.1.1 Condition Register (CR)

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching). The CR is more fully defined in Book I.

Category VLE uses the entire CR, but some comparison operations and all Branch instructions are limited to using CR0–CR3. The full Book I condition register field and logical operations are provided however.

Figure 18. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, CR Field 0 (CR0) ... CR Field 7 (CR7), which are set by VLE defined instructions in one of the following ways.
- Specified fields of the condition register can be set by a move to the CR from a GPR (mtcrf, mtocrf).
- A specified CR field can be set by a move to the CR from another CR field (e_mcrf) or from XER32:35 (mcrxr).
- CR field 0 can be set as the implicit result of a fixed-point instruction.
- A specified CR field can be set as the result of a fixed-point compare instruction.
- CR field 0 can be set as the result of a fixed-point bit test instruction.

CR0

Other instructions from implemented categories may also set bits in the CR in the same manner that they would when not in VLE mode.

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which the Rc bit is defined and set, and for e_add2i, e_and2i, and e_and2is, the first three bits of CR field 0 (CR32:34) are set by signed comparison of the result to zero, and the fourth bit of CR field 0 (CR35) is copied from the final state of XERSO. “Result” here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the value placed into the target register in 32-bit mode.

\[
\begin{align*}
\text{if (64-bit mode)} & \quad \text{then } M \leftarrow 0 \\
\text{else } M & \leftarrow 32 \\
\text{if } (\text{target_register})_{M:63} < 0 & \quad \text{then } c \leftarrow 0b100 \\
\text{else if } (\text{target_register})_{M:63} > 0 & \quad \text{then } c \leftarrow 0b010 \\
\text{else} & \quad c \leftarrow 0b001 \\
\text{CR0} & \leftarrow c \parallel XERSO
\end{align*}
\]

If any portion of the result is undefined, the value placed into the first three bits of CR field 0 is undefined.

The bits of CR field 0 are interpreted as shown below.

<table>
<thead>
<tr>
<th>CR Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Negative (LT) The result is negative.</td>
</tr>
<tr>
<td>33</td>
<td>Positive (GT) The result is positive.</td>
</tr>
<tr>
<td>34</td>
<td>Zero (EQ) The result is 0.</td>
</tr>
<tr>
<td>35</td>
<td>Summary overflow (SO) This is a copy of the contents of XERSO at the completion of the instruction.</td>
</tr>
</tbody>
</table>

4.1.1.1 Condition Register Setting for Compare Instructions

For compare instructions, a CR field specified by the BF operand for the e_cmph, e_cmphl, e_cmpi, and e_cmpli instructions, or CR0 for the se_cmpl, e_cmph16i, e_cmphl16i, e_cmpl16i, se_cmph, se_cmphl, se_cmpl, and se_cmpli instructions, is set to reflect the result of the comparison. The CR field bits are interpreted as shown below. A complete description of how the bits are set is...
given in the instruction descriptions and Section 5.6, "Fixed-Point Compare and Bit Test Instructions".

Condition register bits settings for compare instructions are interpreted as follows. (Note: e_cmpl, and e_cmpli instructions have a BF32 field instead of BF field; for these instructions, BF32 should be substituted for BF in the list below.)

**CR Bit Description**

4×BF + 32

*Less Than* (LT)
- For signed fixed-point compare, (RA) or (RX) < sc18, SI, (RB), or (RY).
- For unsigned fixed-point compare, (RA) or (RX) <u sc18, UI, UI5, (RB), or (RY).

4×BF + 33

*Greater Than* (GT)
- For signed fixed-point compare, (RA) or (RX) > sc18, SI, (RB), or (RY).
- For unsigned fixed-point compare, (RA) or (RX) >u sc18, UI, UI5, (RB), or (RY).

4×BF + 34

*Equal* (EQ)
- For fixed-point compare, (RA) or (RX) = sc18, UI, UI5, SI, (RB), or (RY).

4×BF + 35

*Summary Overflow* (SO)
- For fixed-point compare, this is a copy of the contents of XERSO at the completion of the instruction.

**4.1.1.2 Condition Register Setting for the Bit Test Instruction**

The Bit Test Immediate instruction, se_btsti, also sets CR field 0. See the instruction description and also Section 5.6, "Fixed-Point Compare and Bit Test Instructions".

**4.1.2 Link Register (LR)**

VLE instructions use the Link Register (LR) as defined in Book I, although category VLE defines a subset of all variants of Book I conditional branches involving the LR.

**4.1.3 Count Register (CTR)**

VLE instructions use the Count Register (CTR) as defined in Book I, although category VLE defines a subset of the variants of Book I conditional branches involving the CTR.
4.2 Branch Instructions

The sequence of instruction execution can be changed by the branch instructions. Because VLE instructions must be aligned on half-word boundaries, the low-order bit of the generated branch target address is forced to 0 by the processor in performing the branch.

The branch instructions compute the EA of the target in one of the following ways, as described in Section 2.2, "Instruction Storage Addressing Modes".

1. Adding a displacement to the address of the branch instruction.
2. Using the address contained in the LR (Branch to Link Register [and Link]).
3. Using the address contained in the CTR (Branch to Count Register [and Link]).

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK = 1), the EA of the instruction following the branch instruction is placed into the LR after the branch target address has been computed; this is done regardless of whether the branch is taken.

In branch conditional instructions, the BI32 or BI16 instruction field specifies the CR bit to be tested. For 32-bit instructions using BI32, CR32:47 (corresponding to bits in CR0:CR3) may be specified. For 16-bit instructions using BI16, only CR32:35 (bits within CR0) may be specified.

In branch conditional instructions, the BO32 or BO16 field specifies the conditions under which the branch is taken and how the branch is affected by or affects the CR and CTR. Note that VLE instructions also have different encodings for the BO32 and BO16 fields than in Book I's BO field.

If the BO32 field specifies that the CTR is to be decremented, in 64-bit mode CTR0:63 are decremented, and in 32-bit mode CTR32:63 are decremented. If BO16 or BO32 specifies a condition that must be TRUE or FALSE, that condition is obtained from the contents of CRBI32+32 or CRBI16+32. (Note that CR bits are numbered 32:63. BI32 or BI16 refers to the condition register bit field in the branch instruction encoding. For example, specifying BI32 = 2 refers to CR34.)

Encodings for the BO32 field for VLE are shown in Figure 19.

<table>
<thead>
<tr>
<th>BO32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Branch if the condition is false.</td>
</tr>
<tr>
<td>01</td>
<td>Branch if the condition is true.</td>
</tr>
<tr>
<td>10</td>
<td>Decrement CTR_{M,63}, then branch if the decremented CTR_{M,63} ≠ 0</td>
</tr>
<tr>
<td>11</td>
<td>Decrement CTR_{M,63}, then branch if the decremented CTR_{M,63} = 0.</td>
</tr>
</tbody>
</table>

Figure 19. BO32 field encodings

Encodings for the BO16 field for VLE are shown in Figure 20.

<table>
<thead>
<tr>
<th>BO16</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Branch if the condition is false.</td>
</tr>
<tr>
<td>1</td>
<td>Branch if the condition is true.</td>
</tr>
</tbody>
</table>

Figure 20. BO16 field encodings
**Branch [and Link] BD24-form**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_b</td>
<td>target_addr</td>
<td>0</td>
</tr>
<tr>
<td>e_bl</td>
<td>target_addr</td>
<td>1</td>
</tr>
</tbody>
</table>

NIA $\leftarrow_{\text{iens}} \text{CIA + EXTS (BD24 || 0b0)}$

if LK then LR $\leftarrow_{\text{iens}} \text{CIA + 4}$

**Branch [and Link] BD8-form**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>se_b</td>
<td>target_addr</td>
<td>0</td>
</tr>
<tr>
<td>se_bl</td>
<td>target_addr</td>
<td>1</td>
</tr>
</tbody>
</table>

NIA $\leftarrow_{\text{iens}} \text{CIA + EXTS (BD8 || 0b0)}$

if LK then LR $\leftarrow_{\text{iens}} \text{CIA + 2}$

**target_addr** specifies the branch target address.

The branch target address is the sum of BD24 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

LR (if LK=1)

**Branch Conditional [and Link] BD15-form**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_bc</td>
<td>BO32, BI32, target_addr</td>
<td>0</td>
</tr>
<tr>
<td>e_bcl</td>
<td>BO32, BI32, target_addr</td>
<td>1</td>
</tr>
</tbody>
</table>

if (64-bit mode)

then M $\leftarrow$ 0

else M $\leftarrow$ 32

if BO32 then CTR$_{[63]} \leftarrow$ CTR$_{[63]} - 1$

ctr_ok $\leftarrow$ $\neg$BO32 && ($\neg$CTR$_{[63]} \neq 0$) $\oplus$ BO32$_{[1]}$

cond_ok $\leftarrow$ BO32$_{[1]}$ || (CRBI32+32 $\equiv$ BO32$_{[1]}$)

if ctr_ok $\&$ cond_ok then

NIA $\leftarrow_{\text{iens}} \text{CIA + EXTS (BD15 || 0b0)}$

else

NIA $\leftarrow_{\text{iens}} \text{CIA + 4}$

if LK then LR $\leftarrow_{\text{iens}} \text{CIA + 4}$

The BI32 field specifies the Condition Register bit to be tested. The BO32 field is used to resolve the branch as described in Figure 19. **target_addr** specifies the branch target address.

The branch target address is the sum of BD15 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

**Special Registers Altered:**

CTR (if BO32$_{[1]}$=1)

LR (if LK=1)

**Branch Conditional Short Form BD8-form**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>se_bc</td>
<td>BO16, BI16, target_addr</td>
</tr>
</tbody>
</table>

cond_ok $\leftarrow (\text{CRBI16+32} = \text{BO16})$

if cond_ok then

NIA $\leftarrow_{\text{iens}} \text{CIA + EXTS (BD8 || 0b0)}$

else

NIA $\leftarrow_{\text{iens}} \text{CIA + 2}$

The BI16 field specifies the Condition Register bit to be tested. The BO16 field is used to resolve the branch as described in Figure 20. **target_addr** specifies the branch target address.

The branch target address is the sum of BD8 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

**Special Registers Altered:**

None
Branch to Count Register [and Link] C-form

se_bctr (LK=0)
se_bctrl (LK=1)

NIA ← lea CTR0:62 || 0b0
if LK then LR ← lea CIA + 2

The branch target address is CTR0:62 || 0b0 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:
LR (if LK=1)

Branch to Link Register [and Link] C-form

se_blr (LK=0)
se_brlr (LK=1)

NIA ← lea LR0:62 || 0b0
if LK then LR ← lea CIA + 2

The branch target address is LR0:62 || 0b0 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:
LR (if LK=1)
4.3 System Linkage Instructions

The System Linkage instructions enable the program to call upon the system to perform a service and provide a means by which the system can return from performing a service or from processing an interrupt. System Linkage instructions defined by the VLE category are identical in semantics to System Linkage instructions defined in Book I and Book III-E with the exception of the LEV field, but are encoded differently.

**se_sc** provides the same functionality as the Book I (and Book III-E) instruction *sc* without the LEV field. **se_rfi, se_rfci, se_rfdi,** and **se_rfmci** provide the same functionality as the Book III-E instructions *rfi, rfci, rfdi,* and *rfmci* respectively.

---

**System Call**  
**C-form,ESC-form**

<table>
<thead>
<tr>
<th>se_sc</th>
<th>02 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEV [Category:Embedded.Hypervisor]</td>
<td></td>
</tr>
</tbody>
</table>

**e_sc**  

- ELEV 36

- if ELEV = 1 then
  - MSR = ELEV

  //
  31 6 11 16 21 31

- If MSRGS = 0 or if ELEV = 1, the effective address of the instruction following the System Call instruction is placed into SRR0 and the contents of the MSR are copied into SRR1. Otherwise, the effective address of the instruction following the System Call instruction is placed into GSRR0 and the contents of the MSR are copied into GSRR1. ELEV values greater than 1 are reserved. Bits 0:3 of the ELEV field (instruction bits 16:19) are treated as a reserved field.

- If ELEV=0, a System Call interrupt is generated. If ELEV=1, an Embedded Hypervisor System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 7.6.10 and Section 7.6.30 of Book III-E.

- If MSRGS = 0 or *se_sc* is executed, and the processor is in guest state, instruction execution resumes at the address given by one of the following.
  - GIVPR0:47 || GIVOR8:59||0b0000 if IVORS [Category:Embedded.Phased-Out] are supported.
  - GIVPR0:51||0x120 if Interrupt Fixed Offsets [Category:Embedded.Phased-In] are supported.

- If MSRGS = 0 or *se_sc* is executed, and the processor is in hypervisor state, instruction execution resumes at the address given by one of the following.
  - IVPR0:47 || IVOR8:59||0b0000 if IVORS [Category:Embedded.Phased-Out] are supported.
  - IVPR0:51||0x120 if Interrupt Fixed Offsets [Category:Embedded.Phased-In] are supported.

- If ELEV=1, the interrupt causes instruction execution to resume at the address given by one of the following.
  - GIVPR0:47 || GIVOR8:59||0b0000 if IVORS [Category:Embedded.Phased-Out] are supported.
  - GIVPR0:51||0x300 if Interrupt Fixed Offsets [Category:Embedded.Phased-In] are supported.

This instruction is context synchronizing.

**Special Registers Altered:**  
SRR0 GSRR0 SRR1 GSRR1 MSR

---

**Programming Note**

*e_sc* serves as both a basic and an extended mnemonic. The Assembler will recognize an *e_sc* mnemonic with one operand as the basic form, and an *e_sc* mnemonic with no operand as the extended form. In the extended form, the ELEV operand is omitted and assumed to be 0.
**Illegal**

\[se\text{\_illega}l\]

\[\begin{array}{c}
<table>
<thead>
<tr>
<th>\text{Code}</th>
<th>\text{Size}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>
\end{array}\]

\textit{se\_illegal} is used to request an Illegal Instruction exception.

The behavior is the same as if an illegal instruction was executed.

This instruction is context synchronizing.

**Special Registers Altered:**
- SRR0
- SRR1
- MSR
- ESR

---

**Return From Machine Check Interrupt**

**C-form**

\[se\text{\_rfmci}\]

\[\begin{array}{c}
<table>
<thead>
<tr>
<th>\text{Code}</th>
<th>\text{Size}</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>
\end{array}\]

The \textit{se\_rfmci} instruction is used to return from a machine check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address MCSRR0\_0:62||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 7 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in MCSRR0 at the time of the execution of the \textit{se\_rfmci}).

This instruction is privileged and context synchronizing.

**Special Registers Altered:**
- MSR
## Return From Critical Interrupt C-form

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>se_rfci</code></td>
<td>The <code>se_rfci</code> instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously. The contents of CSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address CSRR0_0.62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>se_rfi</code></td>
<td>The <code>se_rfi</code> instruction is used to return from a non-critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously. The contents of SRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched under control of the new MSR value from the address SRR0_0.62</td>
</tr>
</tbody>
</table>
The `se_rfdi` instruction is used to return from a debug class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address DSRR0:62||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the save/restore registers by the interrupt processing mechanism (see Chapter 7 of Book III-E) is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in DSRR0 at the time of the execution of `se_rfdi`).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Corequisite Categories:
Embedded. Enhanced Debug

---

The `se_rfgi` instruction is used to return from a guest state base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of Guest Save/Restore Register 1 are placed into the MSR. If the `se_rfgi` is executed in the guest supervisor state (MSR GS PR = 0b10), the bits MSR GS WE are not modified and the bits MSR UCLE DE PMM are modified only if the associated bits in the Machine State Register Protect (MSRP) Register are set to 0. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address GSRR0:62||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the associated save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in GSRR0 at the time of the execution of `se_rfgi`).

This instruction is privileged and context synchronizing.

Special Registers Altered:
MSR

Corequisite Categories:
Embedded. Hypervisor
4.4 Condition Register Instructions

Condition Register instructions are provided to transfer values to and from various portions of the CR. Category VLE does not introduce any additional functionality beyond that defined in Book I for CR operations, but does remap the CR-logical and mcrf instruction functionality into primary opcode 31. These instructions operate identically to the Book I instructions, but are encoded differently.

### Condition Register AND XL-form

**e_crand**  
BT, BA, BB

<table>
<thead>
<tr>
<th>31</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>257</th>
</tr>
</thead>
</table>

\[ \text{CR}_{\text{BT+32}} \leftarrow \text{CR}_{\text{BA+32}} \& \text{CR}_{\text{BB+32}} \]

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ \text{CR}_{\text{BT+32}} \]

### Condition Register AND with Complement XL-form

**e_crandc**  
BT, BA, BB

<table>
<thead>
<tr>
<th>31</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>129</th>
</tr>
</thead>
</table>

\[ \text{CR}_{\text{BT+32}} \leftarrow \text{CR}_{\text{BA+32}} \& \neg \text{CR}_{\text{BB+32}} \]

The bit in the Condition Register specified by BA+32 is ANDed with the one's complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ \text{CR}_{\text{BT+32}} \]

### Condition Register Equivalent XL-form

**e_creqv**  
BT, BA, BB

<table>
<thead>
<tr>
<th>31</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>289</th>
</tr>
</thead>
</table>

\[ \text{CR}_{\text{BT+32}} \leftarrow \text{CR}_{\text{BA+32}} \equiv \text{CR}_{\text{BB+32}} \]

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ \text{CR}_{\text{BT+32}} \]

### Condition Register NAND XL-form

**e_crnand**  
BT, BA, BB

<table>
<thead>
<tr>
<th>31</th>
<th>BT</th>
<th>BA</th>
<th>BB</th>
<th>225</th>
</tr>
</thead>
</table>

\[ \text{CR}_{\text{BT+32}} \leftarrow \neg (\text{CR}_{\text{BA+32}} \& \text{CR}_{\text{BB+32}}) \]

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**

\[ \text{CR}_{\text{BT+32}} \]
**Condition Register NOR**  
**XL-form**

\[
e_{\text{cnor}} \quad \text{BT,BA,BB}
\]

\[
\begin{array}{cccccc}
31 & BT & BA & BB & 33 \\
\end{array}
\]

\[
\text{CR}_{BT+32} \leftarrow \neg (\text{CR}_{BA+32} \mid \text{CR}_{BB+32})
\]

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**  
\[\text{CR}_{BT+32}\]

**Condition Register OR with Complement**  
**XL-form**

\[
e_{\text{corc}} \quad \text{BT,BA,BB}
\]

\[
\begin{array}{cccccc}
31 & BT & BA & BB & 417 \\
\end{array}
\]

\[
\text{CR}_{BT+32} \leftarrow \text{CR}_{BA+32} \mid \neg \text{CR}_{BB+32}
\]

The bit in the Condition Register specified by BA+32 is ORed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

**Special Registers Altered:**  
\[\text{CR}_{BT+32}\]

**Move CR Field**  
**XL-form**

\[
e_{\text{mcr}} \quad \text{BF,BFA}
\]

\[
\begin{array}{cccccc}
31 & BF & /// & BFA & /// & 16 \\
\end{array}
\]

\[
\text{CR}_{4x\text{BF}+32;4x\text{BF}+35} \leftarrow \text{CR}_{4x\text{BFA}+32;4x\text{BFA}+35}
\]

The contents of Condition Register field BFA are copied to Condition Register field BF.

**Special Registers Altered:**  
\[\text{CR field BF}\]
Chapter 5. Fixed-Point Instructions

This section lists the fixed-point instructions supported by category VLE.

5.1 Fixed-Point Load Instructions

The fixed-point Load instructions compute the effective address (EA) of the memory to be accessed as described in Section 2.1, “Data Storage Addressing Modes”.

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into RT or RZ.

Category VLE supports both Big- and Little-Endian byte ordering for data accesses.

Some fixed-point load instructions have an update form in which RA is updated with the EA. For these forms, if RA≠0 and RA≠RT, the EA is placed into RA and the memory element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT. If RA=0 or RA =RT, the instruction form is invalid. This is the same behavior as specified for load with update instructions in Book I.

The fixed-point Load instructions from Book I, lbzx, lhzx, lhzx, lhax, lhau, lwzx, and lwzux are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.2 of Book I for the instruction definitions.

The fixed-point Load instructions from Book I, lwax, lwaux, ldx, and ldux are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.2 of Book I for the instruction definitions.
### Load Byte and Zero D-form

- **Syntax:**
  
  ```
  e_lbz     RT,D(RA)
  ```

- **Operands:**
  
  - `RT`: Destination register
  - `D`: Immediate data
  - `RA`: Source register

- **Instructions:**
  
  ```
  12  RT  RA  D  31
  ```

- **Operation:**
  
  - If RA = 0 then b ← 0
  - Else b ← {RA}
  - EA ← b + EXTS(D)
  - RT ← 00 || MEM(EA, 1)

- **Effect:**
  
  Let the effective address (EA) be the sum (RA0) + D. The byte in storage addressed by EA is loaded into RT56:63. RT0:55 are set to 0.

- **Special Registers Altered:**
  
  None

### Load Halfword Algebraic D-form

- **Syntax:**
  
  ```
  e_lha     RT,D(RA)
  ```

- **Operands:**
  
  - `RT`: Destination register
  - `D`: Immediate data
  - `RA`: Source register

- **Instructions:**
  
  ```
  06  RT  RA  0  D8  31
  ```

- **Operation:**
  
  ```
  EA ← (RA) + EXTS(D8)
  RT ← 00 || MEM(EA, 1)
  RA ← EA
  ```

- **Effect:**
  
  Let the effective address (EA) be the sum (RA) + D8. The byte in storage addressed by EA is loaded into RT56:63. RT0:55 are set to 0.
  
  EA is placed into register RA.

- **Special Registers Altered:**
  
  None

### Load Byte and Zero Short Form SD4-form

- **Syntax:**
  
  ```
  se_lbz    RZ,SD4(RX)
  ```

- **Operands:**
  
  - `RZ`: Destination register
  - `SD4`: Immediate data
  - `RX`: Source register

- **Instructions:**
  
  ```
  08  SD4  RZ  RX  15
  ```

- **Operation:**
  
  ```
  EA ← (RX) + 00 || SD4
  RZ ← 00 || MEM(EA, 1)
  ```

- **Effect:**
  
  Let the effective address (EA) be the sum RX + SD4. The byte in storage addressed by EA is loaded into RT56:63. RT0:55 are set to 0.

- **Special Registers Altered:**
  
  None

### Load Halfword Algebraic Short Form SD4-form

- **Syntax:**
  
  ```
  se_lhz    RZ,SD4(RX)
  ```

- **Operands:**
  
  - `RZ`: Destination register
  - `SD4`: Immediate data
  - `RX`: Source register

- **Instructions:**
  
  ```
  10  SD4  RZ  RX  15
  ```

- **Operation:**
  
  ```
  EA ← (RX) + 00 || SD4
  RZ ← 00 || MEM(EA, 1)
  ```

- **Effect:**
  
  Let the effective address (EA) be the sum RX + (SD4 || 0). The halfword in storage addressed by EA is loaded into RZ48:63. RZ0:47 are set to 0.

- **Special Registers Altered:**
  
  None

---

**Version 2.07**

**Power ISA™ - Book VLE**

**1286**
Load Halfword Algebraic with Update

\[ e_{lhau} \quad RT,D8(RA) \]

\[ \begin{array}{cccccc}
0 & 6 & RT & RA & 0 & 3 & D8 \\
\end{array} \]

\[ \begin{align*}
EA & \leftarrow (RA) + \text{EXTS}(D8) \\
RT & \leftarrow \text{EXTS} (\text{MEM}(EA, 2)) \\
RA & \leftarrow EA
\end{align*} \]

Let the effective address (EA) be the sum (RA) + D8. The halfword in storage addressed by EA is loaded into RT_{48:63}. RT_{0:47} are filled with a copy of bit 0 of the loaded halfword.

EA is placed into RA.

If RA = 0 or RA = RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword and Zero with Update

\[ e_{lhzu} \quad RT,D8(RA) \]

\[ \begin{array}{cccccc}
0 & 6 & RT & RA & 0 & 1 & D8 \\
\end{array} \]

\[ \begin{align*}
EA & \leftarrow (RA) + \text{EXTS}(D8) \\
RT & \leftarrow 00 \| \text{MEM}(EA, 2) \\
RA & \leftarrow EA
\end{align*} \]

Let the effective address (EA) be the sum (RA) + D8. The halfword in storage addressed by EA is loaded into RT_{48:63}. RT_{0:47} are set to 0.

EA is placed into register RA.

If RA = 0 or RA = RT, the instruction form is invalid.

Special Registers Altered:
None

Load Word and Zero

\[ e_{lwz} \quad RT,D(RA) \]

\[ \begin{array}{cccc}
20 & RT & RA \\
\end{array} \]

if RA = 0 then \[ b \leftarrow 0 \]
else \[ b \leftarrow (RA) \]

\[ EA \leftarrow b + \text{EXTS}(D) \\
RT \leftarrow 00 \| \text{MEM}(EA, 4) \]

Let the effective address (EA) be the sum (RA|0) + D. The word in storage addressed by EA is loaded into RT_{32:63}. RT_{0:31} are set to 0.

Special Registers Altered:
None

Load Word and Zero Short Form

\[ se_{lwz} \quad RZ,SD4(RX) \]

\[ \begin{array}{cccc}
12 & SD4 & RZ & RX \\
\end{array} \]

\[ \begin{align*}
EA & \leftarrow (RX) + (SD4 \| 00) \\
RZ & \leftarrow 00 \| \text{MEM}(EA, 2)
\end{align*} \]

Let the effective address (EA) be the sum (RX) + (SD4 \| 00). The word in storage addressed by EA is loaded into RZ_{32:63}. RZ_{0:31} are set to 0.

Special Registers Altered:
None
Load Word and Zero with Update D8-form

e_lwzu      RT,D8(RA)

Let the effective address (EA) be the sum (RA) + D8. The word in storage addressed by EA is loaded into RT32:63. RT0:31 are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None
5.2 Fixed-Point Store Instructions

The fixed-point Store instructions compute the EA of the memory to be accessed as described in Section 2.1, “Data Storage Addressing Modes”.

The contents of register RS or RZ are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Category VLE supports both Big- and Little-Endian byte ordering for data accesses.

Some fixed-point store instructions have an update form, in which register RA is updated with the effective address. For these forms, the following rules (from Book I) apply.

- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target memory element and then EA is placed into register RA (RS).

The fixed-point Store instructions from Book I, stbx, stbux, sthx, sthux, stwx, and stwux are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.3 of Book I for the instruction definitions.

The fixed-point Store instructions from Book I, stdx and stdux are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.3 of Book I for the instruction definitions.

### Store Byte

<table>
<thead>
<tr>
<th>D-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_stb</td>
</tr>
</tbody>
</table>

if RA = 0 then b ← 0
else b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 1) ← (RS)_{56:63}

Let the effective address (EA) be the sum (RAi0)+ D. (RS)_{56:63} are stored in the byte in storage addressed by EA.

**Special Registers Altered:**

None

### Store Byte Short Form

<table>
<thead>
<tr>
<th>D4-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>se_stb</td>
</tr>
</tbody>
</table>

EA ← (RX) + EXTS(SD4)
MEM(EA, 1) ← (RZ)_{56:63}

Let the effective address (EA) be the sum (RX) + SD4. (RZ)_{56:63} are stored in the byte in storage addressed by EA.

**Special Registers Altered:**

None
Store Byte with Update\hspace{1cm}D8-form

\textbf{e\textunderscore stbu } RS,D8(RA)

\begin{tabular}{cccccc}
   & 06 & RS & RA & 04 & D8 \\
\hline
\end{tabular}

\begin{align*}
EA & \leftarrow (RA) + \text{EXTS}(D8) \\
\text{MEM}(EA, 1) & \leftarrow (RS)_{56:63} \\
RA & \leftarrow EA
\end{align*}

Let the effective address (EA) be the sum (RA) + D8. (RS)_{56:63} are stored in the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Halfword\hspace{1cm}D-form

\textbf{e\textunderscore sth } RS,D(RA)

\begin{tabular}{cccc}
   & 23 & RS & RA & D \\
\hline
\end{tabular}

\begin{align*}
\text{if } RA = 0 & \text{ then } b \leftarrow 0 \\
\text{else } & b \leftarrow (RA) \\
EA & \leftarrow b + \text{EXTS}(D) \\
\text{MEM}(EA, 2) & \leftarrow (RS)_{48:63}
\end{align*}

Let the effective address (EA) be the sum (RA|0) + D. (RS)_{48:63} are stored in the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Halfword Short Form\hspace{1cm}SD4-form

\textbf{se\textunderscore sth } RZ,SD4(RX)

\begin{tabular}{cccc}
   & 11 & SD4 & RZ & RX \\
\hline
\end{tabular}

\begin{align*}
EA & \leftarrow (RX) + (^{23}_{0} \| \text{SD4} \| 0) \\
\text{MEM}(EA, 2) & \leftarrow (RZ)_{48:63}
\end{align*}

Let the effective address (EA) be the sum (RX) + (SD4 \| 0). (RZ)_{48:63} are stored in the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Halfword with Update\hspace{1cm}D8-form

\textbf{e\textunderscore sthu } RS,D8(RA)

\begin{tabular}{cccccc}
   & 06 & RS & RA & 05 & D8 \\
\hline
\end{tabular}

\begin{align*}
EA & \leftarrow (RA) + \text{EXTS}(D8) \\
\text{MEM}(EA, 2) & \leftarrow (RS)_{48:63} \\
RA & \leftarrow EA
\end{align*}

Let the effective address (EA) be the sum (RA) + D8. (RS)_{48:63} are stored in the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None
### Store Word

**D-form**

<table>
<thead>
<tr>
<th></th>
<th>21</th>
<th>RS</th>
<th>RA</th>
<th>16</th>
<th>D</th>
</tr>
</thead>
</table>

- **e_stw** RS,D(RA)

  if RA = 0 then b ← 0
  else b ← (RA)
  EA ← b + EXTS(D)
  MEM(EA, 4) ← (RS)_{32:63}

Let the effective address (EA) be the sum (RA)₀ + D. (RS)_{32:63} are stored in the word in storage addressed by EA.

**Special Registers Altered:**

None

### Store Word Short Form

**SD4-form**

<table>
<thead>
<tr>
<th>13</th>
<th>SD4</th>
<th>RZ</th>
<th>RX</th>
</tr>
</thead>
</table>

- **se_stw** RZ,SD4(RX)

  EA ← (RX) + (580 | SD4 | 20)
  MEM(EA, 4) ← (RZ)_{32:63}

Let the effective address (EA) be the sum (RX)₀ + (SD4 | 00). (RZ)_{32:63} are stored in the word in storage addressed by EA.

**Special Registers Altered:**

None

### Store Word with Update

**D8-form**

<table>
<thead>
<tr>
<th>06</th>
<th>06</th>
<th>D8</th>
</tr>
</thead>
</table>

- **e_stwu** RS,D8(RA)

  EA ← (RA) + EXTS(D8)
  MEM(EA, 4) ← (RS)_{32:63}
  RA ← EA

Let the effective address (EA) be the sum (RA)₀ + D8. (RS)_{32:63} are stored in the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

**Special Registers Altered:**

None
5.3 Fixed-Point Load and Store with Byte Reversal Instructions

The fixed-point Load with Byte Reversal and Store with Byte Reversal instructions from Book I, \texttt{lhbrx}, \texttt{lwbrx}, \texttt{sthbrx}, and \texttt{stwbrx} are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.5 of Book I for the instruction definitions.

5.4 Fixed-Point Load and Store Multiple Instructions

The Load/Store Multiple instructions have preferred forms; see Section 1.8.1 of Book I. In the preferred forms storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{Load Multiple Word} & \textbf{D8-form} & \textbf{Store Multiple Word} & \textbf{D8-form} \\
\hline
\texttt{e_lmw} & \texttt{RT,D8(RA)} & \texttt{e_stmw} & \texttt{RS,D8(RA)} \\
\hline
0  & 6 & RT & 11 & RA & 16 & 08 & 24 & D8 \\
0  & 6 & RS & 11 & RA & 16 & 09 & 24 & D8 \\
\hline
\end{tabular}

\[
\begin{align*}
\text{if } RA = 0 & \text{ then } b \leftarrow 0 \\
\text{else} & \quad b \leftarrow \text{(RA)} \\
\text{EA} & \leftarrow b + \text{EXTS(D8)} \\
r & \leftarrow RT \\
\text{do while } r \leq 31 \\
& \quad \text{GPR}(r) \leftarrow \ll_{16} \| \text{MEM}(EA, 4) \\
r & \leftarrow r + 1 \\
\text{EA} & \leftarrow \text{EA} + 4 \\
\end{align*}
\]

Let \( n = (32 - RT) \). Let the effective address (EA) be the sum (RA(0) + D8).

\( n \) consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

If RA is in the range of registers to be loaded, including the case in which RA = 0, the instruction form is invalid.

\textbf{Special Registers Altered:}

None
5.5 Fixed-Point Arithmetic Instructions

The fixed-point Arithmetic instructions use the contents of the GPRs as source operands, and place results into GPRs, into status bits in the XER and into CR0.

The fixed-point Arithmetic instructions treat source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The e_add2i instruction and other Arithmetic instructions with Rc=1 set the first three bits of CR0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to 0. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

e_addi[,] and e_subfi[,] always set CA to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode.

The fixed-point Arithmetic instructions from Book I, add[,], addo[,], addci[,], addco[,], addi[,], addo[,], addmi[,], addmei[,], addmeo[,], addzei[,], addzeo[,], divw[,], divwo[,], divwu[,], mulhw[,], mulhwu[,], mulw[,], mulwo[,], neg[,], nego[,], subf[,], subfo[,], subfe[,], subfco[,], subfze[,], subfzeo[,], subfc[,], and subfco[,] are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.9 of Book I for the instruction definitions.

The fixed-point Arithmetic instructions from Book I, mulld[,], mulldo[,], mulhld[,], mulhdu[,], muld[,], muldou[,], divd[,], divdu[,], and divdou[,] are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to these in Book I; see Section 3.3.9 of Book I for the instruction definitions.
Add Short Form

\textbf{RR-form}

\begin{itemize}
  \item se\_add RX,RY
  \begin{tabular}{cccc}
    01 & 0 & RX & RY
  \end{tabular}
  \end{itemize}

The sum (RX + RY) is placed into register RX.

Special Registers Altered:

\begin{itemize}
  \item None
\end{itemize}

Add Immediate

\textbf{D-form}

\begin{itemize}
  \item e\_add16i RT,RA,SI
  \begin{tabular}{cccc}
    07 & RT & RA & SI
  \end{tabular}
  \end{itemize}

The sum (RA + SI) is placed into register RT.

Special Registers Altered:

\begin{itemize}
  \item None
\end{itemize}

Add Immediate Short Form

\textbf{OIM5-form}

\begin{itemize}
  \item se\_add RX,oimm
  \begin{tabular}{cccc}
    08 & 0 & OIM5 & RX
  \end{tabular}
  \end{itemize}

The sum (RX + oimm) is placed into RX. The value of oimm must be in the range of 1 to 32.

Special Registers Altered:

\begin{itemize}
  \item None
\end{itemize}
Add Scaled Immediate Carrying

SCI8-form

e_addic  RT,RA,sci8  (Rc=0)
e_addic. RT,RA,sci8  (Rc=1)

\[
\begin{array}{cccccccc}
06 & RT & RA & 9 & Rcf & SCL & UI8 \\
\end{array}
\]

sci8 \leftarrow \frac{56}{SCL8F} \ || \ UI8 \ || \frac{SCL8F}{8F}

RT \leftarrow (RA) + sci8

The sum (RA) + sci8 is placed into register RT.

Special Registers Altered:
- CR0 (if Rc=1)
- CA

Subtract

RR-form

se_sub  RX,RY

\[
\begin{array}{cccccccc}
1 & & & & & & \\
2 & RY & RX \\
\end{array}
\]

RX \leftarrow (RX) + \neg(RY) + 1

The sum (RX) + \neg(RY) + 1 is placed into register RX.

Special Registers Altered:
- None

Subtract From Short Form

RR-form

se_subf  RX,RY

\[
\begin{array}{cccccccc}
01 & & & & & & \\
3 & RY & RX \\
\end{array}
\]

RX \leftarrow \neg(RX) + (RY) + 1

The sum \neg(RX) + (RY) + 1 is placed into register RX.

Special Registers Altered:
- None

Subtract From Scaled Immediate Carrying

SCI8-form

e_subfic  RT,RA,sci8  (Rc=0)
e_subfic. RT,RA,sci8  (Rc=1)

\[
\begin{array}{cccccccc}
06 & RT & RA & 9 & Rcf & SCL & UI8 \\
\end{array}
\]

sci8 \leftarrow \frac{56}{SCL8F} \ || \ UI8 \ || \frac{SCL8F}{8F}

RT \leftarrow \neg(RA) + sci8 + 1

The sum \neg(RA) + sci8 + 1 is placed into register RT.

Special Registers Altered:
- CR0 (if Rc=1)
- CA

Subtract Immediate

OIM5-form

se_subi  RX,oimm  (Rc=0)
se_subi. RX,oimm  (Rc=1)

\[
\begin{array}{cccccccc}
09 & & & & & & \\
Rc & OIM5 & RX \\
\end{array}
\]

oimm \leftarrow \frac{59}{} \ || \ OIM5 + 1

RX \leftarrow (RX) + \neg(oimm) + 1

The sum (RA) + \neg(oimm) + 1 is placed into register RX.

The value of oimm must be in the range 1 to 32.

Special Registers Altered:
- CR0 (if Rc=1)
Multiply Low Scaled Immediate SCI8-form

e_mulli RT,RA,sci8

```
  06  RT  RA  20  F  SCL  UI8
```

```
scli8 ← 56-SCL×F || UI8 || SCL×8
prod[0:127] ← (RA) × sci8
RT ← prod[64:127]
```

The 64-bit first operand is (RA). The 64-bit second operand is the sci8 operand. The low-order 64-bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

None

Multiply (2 operand) Low Immediate I16A-form

e_mulli2i RA,si

```
  28  si  RA  20  si
```

```
prod[0:127] ← (RA) × EXTS(si)
RA ← prod[64:127]
```

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the si operand. The low-order 64-bits of the 128-bit product of the operands are placed into register RA.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

None

Multiply Low Word Short Form RR-form

se_mullw RX,RY

```
  01  RX  RY
```

```
RX ← (RX)32:63 × (RY)32:63
```

The 32-bit operands are the low-order 32-bits of RX and of RY. The 64-bit product of the operands is placed into register RX.

Both operands and the product are interpreted as signed integers.

**Special Registers Altered:**

None

Negate Short Form R-form

se_neg RX

```
  03  RX
```

```
RX ← ¬(RX) + 1
```

The sum ¬(RX) + 1 is placed into register RX.

If the processor is in 64-bit mode and register RX contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative 64-bit number. Similarly, if the processor is in 32-bit mode and register RX contains the most negative 32-bit number (0x8000_0000), the result is the most negative 32-bit number.

**Special Registers Altered:**

None
5.6 Fixed-Point Compare and Bit Test Instructions

The fixed-point Compare instructions compare the contents of register RA or register RX with one of the following:

- The value of the scaled immediate field $sci_8$ formed from the F, UI8, and SCL fields as:
  
  $sci_8 = 56 \times SCL \times F \ || \ UI8 \ || SCL \times F$

- The zero-extended value of the UI field
- The zero-extended value of the UI5 field
- The sign-extended value of the SI field
- The contents of register RB or register RY.

The following comparisons are signed: $e_{cmph}$, $e_{cmp}$, $e_{cmp16i}$, $e_{cmph16i}$, $se_{cmp}$, $se_{cmph}$, and $se_{cmp16}$.

The following comparisons are unsigned: $e_{cmpl}$, $e_{cmpli}$, $e_{cmpl16i}$, $e_{cmpl16i}$, $se_{cmpl}$, $se_{cmpli}$, and $se_{cmpl16}$.

The fixed-point Bit Test instruction tests the bit specified by the UI5 instruction field and sets the CR0 field as follows.

<table>
<thead>
<tr>
<th>Bit Name Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 LT</td>
<td>Always set to 0</td>
</tr>
<tr>
<td>1 GT</td>
<td>RX_{ui5} = 1</td>
</tr>
<tr>
<td>2 EQ</td>
<td>RX_{ui5} = 0</td>
</tr>
<tr>
<td>3 SO</td>
<td>Summary overflow from the XER</td>
</tr>
</tbody>
</table>

The fixed-point Compare instructions from Book I, $cmp$ and $cmpl$ are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.10 of Book I for the instruction definitions.

### Bit Test Immediate

**IM5-form**

<table>
<thead>
<tr>
<th>se_BTSTI</th>
<th>RX,UI5</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

- $a \leftarrow UI5$
- $b \leftarrow a^{31:0} || 0^{31:0}$
- $c \leftarrow (RX) \ & \ b$
- if $c = 0$ then $d \leftarrow 0b001$ else $d \leftarrow 0b101$
- $CR0 \leftarrow d \ || XER_{SO}$

Bit UI5+32 of register RX is tested for equality to '0' and the result is recorded in CR0. EQ is set if the tested bit is 0, LT is cleared, and GT is set to the inverse value of EQ.

**Special Registers Altered:**

- CR0

### Compare Immediate Word

**I16A-form**

<table>
<thead>
<tr>
<th>e_cmp16i</th>
<th>RA,si</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>si</th>
<th>16</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- $b \leftarrow EXTS(si)$
- if $(RA)_{32:63} < b_{32:63}$ then $c \leftarrow 0b100$
- if $(RA)_{32:63} > b_{32:63}$ then $c \leftarrow 0b010$
- if $(RA)_{32:63} = b_{32:63}$ then $c \leftarrow 0b001$
- $CR0 \leftarrow c \ || XER_{SO}$

The low-order 32 bits of register RA are compared with si, treating operands as signed integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**

- CR0
**Compare Scaled Immediate Word**  
**SCI8-form**

\[ e_{cmpi} \quad BF32, RA, sci8 \]

<table>
<thead>
<tr>
<th></th>
<th>06</th>
<th>0001</th>
<th>BF32</th>
<th>RA</th>
<th>21</th>
<th>F</th>
<th>SCL</th>
<th>UI8</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ sci8 \leftarrow SCL \times 8 \quad UI8 \quad SCL \times 8 \]

if \((RA)_{32:63} < sci8_{32:63}\) then \(c \leftarrow 0b100\)
if \((RA)_{32:63} > sci8_{32:63}\) then \(c \leftarrow 0b010\)
if \((RA)_{32:63} = sci8_{32:63}\) then \(c \leftarrow 0b001\)

CR4×BF32+32×BF32+35 \(\leftarrow c \parallel XERSO\)

The low-order 32 bits of register RA are compared with sci8, treating operands as signed integers. The result of the comparison is placed into CR field BF32.

**Special Registers Altered:**  
CR field BF32

---

**Compare Word**  
**RR-form**

\[ se_{cmp} \quad RX, RY \]

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>0</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

if \((RX)_{32:63} < (RY)_{32:63}\) then \(c \leftarrow 0b100\)
if \((RX)_{32:63} > (RY)_{32:63}\) then \(c \leftarrow 0b010\)
if \((RX)_{32:63} = (RY)_{32:63}\) then \(c \leftarrow 0b001\)

CR0 \(\leftarrow c \parallel XERSO\)

The low-order 32 bits of register RX are compared with the low-order 32 bits of register RY, treating operands as signed integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**  
CR0

---

**Compare Immediate Word Short Form**  
**IM5-form**

\[ se_{cmpi} \quad RX, UI5 \]

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>11</th>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ b \leftarrow 59 || UI5 \]

if \((RX)_{32:63} < UI5_{32:63}\) then \(c \leftarrow 0b100\)
if \((RX)_{32:63} > UI5_{32:63}\) then \(c \leftarrow 0b010\)
if \((RX)_{32:63} = UI5_{32:63}\) then \(c \leftarrow 0b001\)

CR0 \(\leftarrow c \parallel XERSO\)

The low-order 32 bits of register RX are compared with UI5, treating operands as signed integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**  
CR0

---

**Compare Logical Immediate Word**  
**I16A-form**

\[ e_{cmpi16i} \quad RA, ui \]

<table>
<thead>
<tr>
<th></th>
<th>28</th>
<th>21</th>
<th>ui</th>
<th>21</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ b \leftarrow 98 || ui \]

if \((RA)_{32:63} < ui_{32:63}\) then \(c \leftarrow 0b100\)
if \((RA)_{32:63} > ui_{32:63}\) then \(c \leftarrow 0b010\)
if \((RA)_{32:63} = ui_{32:63}\) then \(c \leftarrow 0b001\)

CR0 \(\leftarrow c \parallel XERSO\)

The low-order 32 bits of register RA are compared with ui, treating operands as unsigned integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**  
CR0
**Compare Logical Scaled Immediate Word**

### SCI8-form

```plaintext
e_cmpli BF32,RA,sci8
```

<table>
<thead>
<tr>
<th>06</th>
<th>001</th>
<th>BF32</th>
<th>RA</th>
<th>21</th>
<th>FSCL</th>
<th>UI8</th>
</tr>
</thead>
</table>

`sci8 ← $^{56}$SCL$^{8}$F $||$ UI8 $||$ SCL$^{8}$F`

- if $(RA)_{32:63} < u sci8_{32:63}$ then $c ← 0b100$
- if $(RA)_{32:63} > u sci8_{32:63}$ then $c ← 0b010$
- if $(RA)_{32:63} = sci8_{32:63}$ then $c ← 0b001$

`CR4 × BF32+32:4 × BF32+35 $c || XERSO$`

The low-order 32 bits of register RA are compared with sci8, treating operands as unsigned integers. The result of the comparison is placed into CR field BF32.

**Special Registers Altered:**
- CR field BF32

### RR-form

```plaintext
se_cmpli RX,RY
```

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
</table>

- if $(RX)_{32:63} < u (RY)_{32:63}$ then $c ← 0b100$
- if $(RX)_{32:63} > u (RY)_{32:63}$ then $c ← 0b010$
- if $(RX)_{32:63} = (RY)_{32:63}$ then $c ← 0b001$

`CR0 $c || XERSO$`

The low-order 32 bits of register RX are compared with the low-order 32 bits of register RY, treating operands as unsigned integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**
- CR0

**Compare Logical Immediate Word**

### OIM5-form

```plaintext
se_cmpli RX,oimm
```

<table>
<thead>
<tr>
<th>08</th>
<th>OIM5</th>
<th>RX</th>
</tr>
</thead>
</table>

`oimm ← $^{59}$ OIM5 + 1`

- if $(RX)_{32:63} < u oimm_{32:63}$ then $c ← 0b100$
- if $(RX)_{32:63} > u oimm_{32:63}$ then $c ← 0b010$
- if $(RX)_{32:63} = oimm_{32:63}$ then $c ← 0b001$

`CR0 $c || XERSO$`

The low-order 32 bits of register RX are compared with oimm, treating operands as unsigned integers. The result of the comparison is placed into CR0. The value of oimm must be in the range of 1 to 32.

**Special Registers Altered:**
- CR0

### X-form

```plaintext
e_cmmp BF,RA,RB
```

<table>
<thead>
<tr>
<th>31</th>
<th>BF</th>
<th>0</th>
<th>RA</th>
<th>RB</th>
<th>14</th>
</tr>
</thead>
</table>

- $a ← \text{EXTS}((RA)_{48:63})$
- $b ← \text{EXTS}((RB)_{48:63})$

- if $a < b$ then $c ← 0b100$
- if $a > b$ then $c ← 0b010$
- if $a = b$ then $c ← 0b001$

`CR4 × BF32+32:4 × BF32+35 $c || XERSO$`

The low-order 16 bits of register RA are compared with the low-order 16 bits of register RB, treating operands as signed integers. The result of the comparison is placed into CR field BF.

**Special Registers Altered:**
- CR field BF
**Compare Halfword Short Form**  
**RR-form**

```
se_cmph RX,RY
```

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>←</td>
<td>EXTS((RX)_{48:63})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>←</td>
<td>EXTS((RY)_{48:63})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a &lt; b then</td>
<td>c</td>
<td>←</td>
<td>0b100</td>
</tr>
<tr>
<td>if</td>
<td>a &gt; b then</td>
<td>c</td>
<td>←</td>
<td>0b010</td>
</tr>
<tr>
<td>if</td>
<td>a = b then</td>
<td>c</td>
<td>←</td>
<td>0b001</td>
</tr>
<tr>
<td>CR0</td>
<td>←</td>
<td>c</td>
<td></td>
<td>XER_{SO}</td>
</tr>
</tbody>
</table>

The low-order 16 bits of register RX are compared with the low-order 16 bits of register RY, treating operands as signed integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**

CR0

**Compare Halfword Immediate**  
**I16A-form**

```
e_cmph16i RA,si
```

<table>
<thead>
<tr>
<th></th>
<th>28</th>
<th>si</th>
<th>RA</th>
<th>22</th>
<th>si</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>←</td>
<td>EXTS((RA)_{48:63})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>←</td>
<td>EXTS(si)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a &lt; b then</td>
<td>c</td>
<td>←</td>
<td>0b100</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a &gt; b then</td>
<td>c</td>
<td>←</td>
<td>0b010</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a = b then</td>
<td>c</td>
<td>←</td>
<td>0b001</td>
<td></td>
</tr>
<tr>
<td>CR0</td>
<td>←</td>
<td>c</td>
<td></td>
<td>XER_{SO}</td>
<td></td>
</tr>
</tbody>
</table>

The low-order 16 bits of register RA are compared with si, treating operands as signed integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**

CR0

**Compare Halfword Logical**  
**X-form**

```
e_cmphl BF,RA,RB
```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>BF</th>
<th>0</th>
<th>RA</th>
<th>16</th>
<th>RB</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>←</td>
<td>EXTZ((RA)_{48:63})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>←</td>
<td>EXTZ((RB)_{48:63})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a &lt;u b then</td>
<td>c</td>
<td>←</td>
<td>0b100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a &gt;u b then</td>
<td>c</td>
<td>←</td>
<td>0b010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a = b then</td>
<td>c</td>
<td>←</td>
<td>0b001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR4 \times BF+32:4 \times BF+35</td>
<td>←</td>
<td>c</td>
<td></td>
<td>XER_{SO}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The low-order 16 bits of register RA are compared with the low-order 16 bits of register RB, treating operands as unsigned integers. The result of the comparison is placed into CR field BF.

**Special Registers Altered:**

CR field BF

**Compare Halfword Logical Short Form**  
**RR-form**

```
se_cmphl RX,RY
```

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>←</td>
<td>(RX)_{48:63}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>←</td>
<td>(RY)_{48:63}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>a &lt;u b then</td>
<td>c</td>
<td>←</td>
<td>0b100</td>
</tr>
<tr>
<td>if</td>
<td>a &gt;u b then</td>
<td>c</td>
<td>←</td>
<td>0b010</td>
</tr>
<tr>
<td>if</td>
<td>a = b then</td>
<td>c</td>
<td>←</td>
<td>0b001</td>
</tr>
<tr>
<td>CR0</td>
<td>←</td>
<td>c</td>
<td></td>
<td>XER_{SO}</td>
</tr>
</tbody>
</table>

The low-order 16 bits of register RX are compared with the low-order 16 bits of register RY, treating operands as unsigned integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**

CR0
**Chapter 5. Fixed-Point Instructions**

### 5.7 Fixed-Point Trap Instructions

The fixed-point Trap instruction from Book I, `tw` is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction is identical to that in Book I; see Section 3.3.11 of Book I for the instruction definition.

The fixed-point Trap instruction from Book I, `td` is available while executing in VLE mode on 64-bit implementations. The mnemonic, decoding, and semantics for the `td` instruction are identical to those in Book I; see Section 3.3.11 of Book I for the instruction definitions.

### 5.8 Fixed-Point Select Instruction

The fixed-point Select instruction provides a means to select one of two registers and place the result in a destination register under the control of a predicate value supplied by a CR bit.

The fixed-point Select instruction from Book I, `isel` is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction is identical to that in Book I; see Section of Book I for the instruction definition.

---

**Compare Halfword Logical Immediate**

**I16A-form**

```
e_cmphl16i RA, ui
```

<table>
<thead>
<tr>
<th>28</th>
<th>ui</th>
<th>RA</th>
<th>23</th>
<th>ui</th>
</tr>
</thead>
</table>

\[
a \leftrightarrow 48_0 \parallel (RA)_{48:63}
b \leftrightarrow 48_0 \parallel ui
\]

- if \( a <_u b \) then \( c \leftarrow 0b100 \)
- if \( a >_u b \) then \( c \leftarrow 0b010 \)
- if \( a = b \) then \( c \leftarrow 0b001 \)
- \( CR0 \leftarrow c \parallel XER_{60} \)

The low-order 16 bits of register RA are compared with the `ui` field, treating operands as signed integers. The result of the comparison is placed into CR0.

**Special Registers Altered:**

- CR0
5.9 Fixed-Point Logical, Bit, and Move Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands. The *Bit* instructions manipulate a bit, or create a bit mask, in a register. The *Move* instructions move a register or an immediate value into a register.

The X-form Logical instructions with Rc=1, the SCI8-form Logical instructions with Rc=1, the RR-form Logical instructions with Rc=1, the *e_and2i*, and *e_and2is* instruction set the first three bits of CR field 0 as the arithmetic instructions described in Section 5.5, “Fixed-Point Arithmetic Instructions”. (Also see Section 4.1.1.) The Logical instructions do not change the SO, OV, and CA bits in the XER.

The fixed-point Logical instructions from Book I, *and*, *or*, *xor*, *nand*, *nor*, *eqv*, *andc*, *orc*, *extsb*, *extsh*, *cntlzwr*, and *popcntb* are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.13 of Book I for the instruction definitions.

The fixed-point Logical instructions from Book I, *extsw* and *cntlzd* are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.13 of Book I for the instruction definitions.

### AND (two operand) Immediate I16L-form

**e_and2i**. RT,ui

<table>
<thead>
<tr>
<th>28</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>ui</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

RT ← (RT) & (480 || ui)

The contents of register RT are ANDed with 480 || ui and the result is placed into register RT.

**Special Registers Altered:**

CR0

### AND Scaled Immediate SCI8-form

**e_andi**. RA,RS,sci8 (Rc=0)

<table>
<thead>
<tr>
<th>56</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sci8</td>
<td>56-SCLx8F</td>
<td></td>
<td>UI8</td>
<td></td>
<td>SCLx8F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td>(RS) &amp; sci8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of register RS are ANDed with sci8 and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if Rc=1)

### AND (2 operand) Immediate Shifted I16L-form

**e_and2is**. RT,ui

<table>
<thead>
<tr>
<th>28</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>ui</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
</tr>
</tbody>
</table>

RT ← (RT) & (120 || ui || 160)

The contents of register RT are ANDed with 120 || ui || 160 and the result is placed into register RT.

**Special Registers Altered:**

CR0

### AND Immediate Short Form IM5-form

**se_andi**. RX,UI5

<table>
<thead>
<tr>
<th>11</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>UI5</td>
<td>RX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RX ← (RX) & 590 || UI5

The contents of register RX are ANDed with 590 || UI5 and the result is placed into register RX.

**Special Registers Altered:**

None

---

1302  Power ISA™ - Book VLE
**OR (two operand) Immediate I16L-form**

\[ e_{\text{or2i}} \quad RT,ui \]

\[
\begin{array}{ccccccc}
28 & RT & ui & 24 & ui & 21 & 19 & 16 & 14 & 12 & 10 & 8 & 6 & 4 & 2 & 0
\end{array}
\]

\[ RT \leftarrow (RT) \mid (480 \mid ui) \]

The contents of register RT are ORed with 480 || ui and the result is placed into register RT.

**Special Registers Altered:**

None

**OR (2 operand) Immediate Shifted I16L-form**

\[ e_{\text{or2is}} \quad RT,ui \]

\[
\begin{array}{ccccccc}
28 & RT & ui & 26 & ui & 21 & 19 & 16 & 14 & 12 & 10 & 8 & 6 & 4 & 2 & 0
\end{array}
\]

\[ RT \leftarrow (RT) \mid (320 \mid ui \mid 160) \]

The contents of register RT are ORed with 320 || ui || 160 and the result is placed into register RT.

**Special Registers Altered:**

None

**OR Scaled Immediate SCI8-form**

\[ e_{\text{ori}} \quad RA,RS,sci8 \]

\[ e_{\text{ori.}} \quad RA,RS,sci8 \]

\[
\begin{array}{ccccccccccc}
06 & RS & RA & 13 & Rcf & SCL & UI8 & 31 & 24 & 17 & 11 & 6 & 1
\end{array}
\]

\[ sci8 \leftarrow 56^{-\text{SCLx8F}} \mid UI8 \mid 56\text{SCLx8F} \]

\[ RA \leftarrow (RS) \mid sci8 \]

The contents of register RS are ORed with sci8 and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if Rc=1)

**XOR Scaled Immediate SCI8-form**

\[ e_{\text{xori}} \quad RA,RS,sci8 \]

\[ e_{\text{xori.}} \quad RA,RS,sci8 \]

\[
\begin{array}{ccccccccccc}
06 & RS & RA & 14 & Rcf & SCL & UI8 & 31 & 24 & 17 & 11 & 6 & 1
\end{array}
\]

\[ sci8 \leftarrow 56^{-\text{SCLx8F}} \mid UI8 \mid 56\text{SCLx8F} \]

\[ RA \leftarrow (RS) \oplus sci8 \]

The contents of register RS are XORed with sci8 and the result is placed into register RA.

**Special Registers Altered:**

CR0 (if Rc=1)

**AND Short Form RR-form**

\[ se_{\text{and}} \quad RX,RY \]

\[ se_{\text{and.}} \quad RX,RY \]

\[
\begin{array}{ccccccccccc}
17 & 1 & RX & 15 & 12 & 9 & 6 & 3 & 0
\end{array}
\]

\[ RX \leftarrow (RX) \land (RY) \]

The contents of register RX are ANDed with the contents of register RY and the result is placed into register RX.

**Special Registers Altered:**

CR0 (if Rc=1)

**AND with Complement Short Form RR-form**

\[ se_{\text{andc}} \quad RX,RY \]

\[
\begin{array}{ccccccccccc}
17 & 1 & RX & 15 & 12 & 9 & 6 & 3 & 0
\end{array}
\]

\[ RX \leftarrow (RX) \land \overline{RY} \]

The contents of register RX are ANDed with the complement of the contents of register RY and the result is placed into register RX.

**Special Registers Altered:**

None
OR Short Form

\[
\text{se_or} \quad RX,RY
\]

<table>
<thead>
<tr>
<th>17</th>
<th>0</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
</table>

\(RX \leftarrow (RX) \lor (RY)\)

The contents of register RX are ORed with the contents of register RY and the result is placed into register RX.

Special Registers Altered:
None

RR-form

NOT Short Form

\[
\text{se_not} \quad RX
\]

<table>
<thead>
<tr>
<th>0</th>
<th>02</th>
<th>RX</th>
</tr>
</thead>
</table>

\(RX \leftarrow \neg(RX)\)

The contents of RX are complemented and placed into register RX.

Special Registers Altered:
None

Bit Clear Immediate

\[
\text{se_bclri} \quad RX,UI5
\]

<table>
<thead>
<tr>
<th>24</th>
<th>0</th>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
</table>

\(a \leftarrow UI5\)

\(RX \leftarrow (RX) \& (a+32 \mid | \mid 31-a1)\)

Bit UI5+32 of register RX is set to 0.

Special Registers Altered:
None

IM5-form

Bit Generate Immediate

\[
\text{se_bgeni} \quad RX,UI5
\]

<table>
<thead>
<tr>
<th>24</th>
<th>1</th>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
</table>

\(a \leftarrow UI5\)

\(RX \leftarrow (a+32 \mid | \mid 1 \mid 31-a0)\)

Bit UI5+32 of register RX is set to 1. All other bits in register RX are set to 0.

Special Registers Altered:
None

IM5-form

Bit Mask Generate Immediate

\[
\text{se_bmaski} \quad RX,UI5
\]

<table>
<thead>
<tr>
<th>11</th>
<th>0</th>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
</table>

If \(a = 0\) then \(RX \leftarrow 64\)

else \(RX \leftarrow 64-a0 \mid a1\)

If UI5 is not zero, the low-order UI5 bits are set to 1 in register RX and all other bits in register RX are set to 0. If UI5 is 0, all bits in register RX are set to 1.

Special Registers Altered:
None

IM5-form

Bit Set Immediate

\[
\text{se_bseti} \quad RX,UI5
\]

<table>
<thead>
<tr>
<th>25</th>
<th>0</th>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
</table>

\(a \leftarrow UI5\)

\(RX \leftarrow (RX) \lor (a+32 \mid | \mid 1 \mid 31-a0)\)

Bit UI5+32 of register RX is set to 1.

Special Registers Altered:
None
### Extend Sign Byte Short Form

**R-form**

```
se_extsb RX
```

```
| 0 | 13 | RX |
```

\[ R_X \leftarrow R_X \lls 56 \parallel (R_X)_{56:63} \]

(RX)$_{56:63}$ are placed into RX$_{56:63}$. Bit 56 of register RX is placed into RX$_{0:55}$.

**Special Registers Altered:**

None

### Extend Sign Halfword Short Form

**R-form**

```
se_extsh RX
```

```
| 0 | 15 | RX |
```

\[ R_X \leftarrow R_X \lls 48 \parallel (R_X)_{48:63} \]

(RX)$_{48:63}$ are placed into RX$_{48:63}$. Bit 48 of register RX is placed into RX$_{0:47}$.

**Special Registers Altered:**

None

### Extend Zero Byte

**R-form**

```
se_extzb RX
```

```
| 0 | 12 | RX |
```

\[ R_X \leftarrow R_X \ll 56 \parallel (R_X)_{56:63} \]

(RX)$_{56:63}$ are placed into RX$_{56:63}$. RX$_{0:55}$ are set to 0.

**Special Registers Altered:**

None

### Extend Zero Halfword

**R-form**

```
se_extzh RX
```

```
| 0 | 14 | RX |
```

\[ R_X \leftarrow R_X \ll 48 \parallel (R_X)_{48:63} \]

(RX)$_{48:63}$ are placed into RX$_{48:63}$. RX$_{0:47}$ are set to 0.

**Special Registers Altered:**

None

### Load Immediate LI20-form

```
e_li RT,LI20
```

```
| 28 | RT | li20 | li20 | li20 |
```

\[ R_T \leftarrow \text{exts}(li20_{1:4} \parallel li20_{5,8} \parallel li20_{9,15}) \]

The sign-extended LI20 field is placed into RT.

**Special Registers Altered:**

None

### Load Immediate Short Form IM7-form

```
se_li RX,UI7
```

```
| 09 | UI7 | RX |
```

\[ R_X \leftarrow R_X \lls 57 \parallel UI7 \]

The zero-extended UI7 field is placed into RX.

**Special Registers Altered:**

None

### Load Immediate Shifted I16L-form

```
e_lis RT,ui
```

```
| 28 | RT | ui |
```

\[ R_T \leftarrow R_T \ll 32 \parallel ui \ll 16 \]

The zero-extended value of ui shifted left 16 bits is placed into RT.

**Special Registers Altered:**

None
**Move from Alternate Register**  
*RR-form*

```plaintext
se_mfar  RX,ARY
```

```
<table>
<thead>
<tr>
<th>p</th>
<th>0</th>
<th>3</th>
<th>ARY</th>
<th>RX</th>
</tr>
</thead>
</table>
```

\[ r \leftarrow ARY+8 \]
\[ RX \leftarrow GPR(r) \]

The contents of register ARY+8 are placed into RX. ARY specifies a register in the range R8:R23.

*Special Registers Altered:*

None

---

**Move Register**  
*RR-form*

```plaintext
se_mr  RX,RY
```

```
<table>
<thead>
<tr>
<th>p</th>
<th>0</th>
<th>1</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
</table>
```

\[ RX \leftarrow (RY) \]

The contents of register RY are placed into RX.

*Special Registers Altered:*

None

---

**Move To Alternate Register**  
*RR-form*

```plaintext
se_mtar  ARX,RY
```

```
<table>
<thead>
<tr>
<th>p</th>
<th>0</th>
<th>2</th>
<th>RY</th>
<th>ARX</th>
</tr>
</thead>
</table>
```

\[ r \leftarrow ARX+8 \]
\[ GPR(r) \leftarrow (RY) \]

The contents of register RY are placed into register ARX+8. ARX specifies a register in the range R8:R23.

*Special Registers Altered:*

None
5.10 Fixed-Point Rotate and Shift Instructions

The fixed-point Shift instructions from Book I, \textit{slw[.]}, \textit{srw[.]}, \textit{sraw[.]}, and \textit{sraw[.]} are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.14.2 of Book I for the instruction definitions.

### Rotate Left Word

**X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SRC Registers</th>
<th>RC Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_rlw</td>
<td>RA,RS,RB</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>e_rlw.</td>
<td>RA,RS,RB</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
31 & RS & RA & RB & 280 & Rc \\
\hline
n & (RB) & 59:63 & RA & I & ROTL_{32}((RS)32:63,n) \\
\end{array}
\]

The contents of register RS are rotated \(32\) left the number of bits specified by \((RB)_{59:63}\) and the result is placed into register RA.

**Special Registers Altered:**

- CR0 (if \(Rc=1\))

### Rotate Left Word Immediate

**X-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SRC Registers</th>
<th>RC Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_rlwI</td>
<td>RA,RS,SH</td>
<td>(Rc=0)</td>
</tr>
<tr>
<td>e_rlwI.</td>
<td>RA,RS,SH</td>
<td>(Rc=1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
31 & RS & RA & SH & 312 & Rc \\
\hline
n & SH & RA & \leftarrow & ROTL_{32}((RS)_{32:63},n) \\
\end{array}
\]

The contents of register RS are rotated \(32\) left \(SH\) bits and the result is placed into register RA.

**Special Registers Altered:**

- CR0 (if \(Rc=1\))

### Rotate Left Word Immediate then Mask Insert

**M-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SRC Registers</th>
<th>RC Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_rlwimi</td>
<td>RA,RS,SH,MB,ME</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
29 & RS & RA & SH & MB & ME \\
\hline
n & SH & RA & \leftarrow & ROTL_{32}((RS)_{32:63},n) \\
\end{array}
\]

The contents of register RS are rotated \(32\) left \(SH\) bits. A mask is generated having 1-bits from bit \(MB+32\) through bit \(ME+32\) and 0-bits elsewhere. The rotated data is inserted into register RA under control of the generated mask.

**Special Registers Altered:**

- None

### Rotate Left Word Immediate then AND with Mask

**M-form**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SRC Registers</th>
<th>RC Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_rlwim</td>
<td>RA,RS,SH,MB,ME</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
29 & RS & RA & SH & MB & ME \\
\hline
n & SH & RA & \leftarrow & ROTL_{32}((RS)_{32:63},n) \\
\end{array}
\]

The contents of register RS are rotated \(32\) left \(SH\) bits. A mask is generated having 1-bits from bit \(MB+32\) through bit \(ME+32\) and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RA.

**Special Registers Altered:**

- None
### Shift Left Word Immediate (X-form)

**Instruction:**
- `e_slwi`  
- `e_slwi.`  

**Operands:**
- `RA`, `RS`, `SH`  
- `RA`, `RS`, `SH`

**Format:**
- 31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

- `n`  
- `SH`  
- `m`  
- `RA`  

**Description:**
- The contents of the low-order 32 bits of register `RS` are shifted left `SH` bits. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into `RA 32:63`. `RA 0:31` are set to 0.

**Special Registers Altered:**
- CR0 (if `Rc=1`)

### Shift Left Word Immediate Short Form (IM5-form)

**Instruction:**
- `se_slwi`

**Operands:**
- `RX`, `UI5`

**Format:**
- 27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

- `n`  
- `UI5`  
- `m`  
- `RX`

**Description:**
- The contents of the low-order 32 bits of register `RX` are shifted left `UI5` bits. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into `RX 32:63`. `RX 0:31` are set to 0.

**Special Registers Altered:**
- None

### Shift Left Word (RR-form)

**Instruction:**
- `se_slw`

**Operands:**
- `RX`, `RY`

**Format:**
- 16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

- `n`  
- `RY`  
- `m`  
- `RX`

**Description:**
- The contents of the low-order 32 bits of register `RX` are shifted left the number of bits specified by `(RY)58:63`. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into `RX 32:63`. `RX 0:31` are set to 0. Shift amounts from 32-63 give a zero result.

**Special Registers Altered:**
- None

### Shift Right Algebraic Word Immediate (IM5-form)

**Instruction:**
- `se_srawi`

**Operands:**
- `RX`, `UI5`

**Format:**
- 26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

- `n`  
- `UI5`  
- `m`  
- `RX`

**Description:**
- The contents of the low-order 32 bits of register `RX` are shifted right `UI5` bits. Bits shifted out of position 63 are lost, and bit 32 of `RX` is replicated to fill the vacated positions on the left. Bit 32 of `RX` is replicated to fill `RX 0:31` and the 32-bit result is placed into `RX 32:63`. `CA` is set to 1 if the low-order 32 bits of register `RX` contain a negative value and any 1-bits are shifted out of bit position 63; otherwise `CA` is set to 0. A shift amount of zero causes `RX` to receive `EXTS((RX)32:63)`, and `CA` to be set to 0.

**Special Registers Altered:**
- `CA`
Shift Right Algebraic Word  

**RR-form**

se_sraw RX,RY

<table>
<thead>
<tr>
<th>16</th>
<th>1</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
</table>

\[
n \leftarrow (RY)_{59:63}
\]
\[
r \leftarrow \text{ROTL}_{32}((RX)_{32:63}, 64-n)
\]
\[
\text{if } (RY)_{58} = 0 \text{ then } m \leftarrow \text{MASK}(n+32, 63)
\]
\[
\text{else } \quad m \leftarrow 64_{0}
\]
\[
s \leftarrow (RX)_{32}
\]
\[
RX \leftarrow r \& m \mid (r \& \neg m)_{32:63} \neq 0
\]
\[
CA \leftarrow s \& ((r \& \neg m)_{32,63} \neq 0)
\]

The contents of the low-order 32 bits of register RX are shifted right the number of bits specified by \((RY)_{58:63}\). Bits shifted out of position 63 are lost, and bit 32 of RX is replicated to fill the vacated positions on the left. Bits 32 of RX is replicated to fill RX_{0:31} and the 32-bit result is placed into RX_{32:63}. CA is set to 1 if the low-order 32 bits of register RX contain a negative value and any 1-bits are shifted out of bit position 63; otherwise CA is set to 0. A shift amount of zero causes RX to receive \(\text{EXTS}((RX)_{32:63})\), and CA to be set to 0. Shift amounts from 32-63 give a result of 64 sign bits, and cause CA to receive the sign bit of \((RX)_{32:63}\).

Special Registers Altered:

- CA

---

Shift Right Word Immediate Short Form  

**IM5-form**

se_srwi RX,UI5

<table>
<thead>
<tr>
<th>26</th>
<th>0</th>
<th>UI5</th>
<th>RX</th>
</tr>
</thead>
</table>

\[
n \leftarrow UI5
\]
\[
r \leftarrow \text{ROTL}_{32}((RX)_{32:63}, 64-n)
\]
\[
m \leftarrow \text{MASK}(n+32, 63)
\]
\[
RX \leftarrow r \& m
\]

The contents of the low-order 32 bits of register RX are shifted right UI5 bits. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into RX_{32:63}, RX_{0:31} are set to 0.

Special Registers Altered:

- None

---

Shift Right Word Immediate  

**X-form**

e_srawi RA,RS,SH (Rc=0)
e_srawi RA,RS,SH (Rc=1)

<table>
<thead>
<tr>
<th>568</th>
<th>Rc</th>
</tr>
</thead>
</table>

\[
n \leftarrow SH
\]
\[
r \leftarrow \text{ROTL}_{32}((RS)_{32:63}, 64-n)
\]
\[
m \leftarrow \text{MASK}(n+32, 63)
\]
\[
RA \leftarrow r \& m
\]

The contents of the low-order 32 bits of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into RA_{32:63}, RA_{0:31} are set to 0.

Special Registers Altered:

- CR0 (if Rc=1)

---

Shift Right Word  

**RR-form**

se_srw RX,RY

<table>
<thead>
<tr>
<th>16</th>
<th>0</th>
<th>RY</th>
<th>RX</th>
</tr>
</thead>
</table>

\[
n \leftarrow (RY)_{59:63}
\]
\[
r \leftarrow \text{ROTL}_{32}((RX)_{32:63}, 64-n)
\]
\[
\text{if } (RY)_{58} = 0 \text{ then } m \leftarrow \text{MASK}(n+32, 63)
\]
\[
\text{else } \quad m \leftarrow 64_{0}
\]
\[
s \leftarrow (RX)_{32}
\]
\[
RX \leftarrow r \& m \mid (r \& \neg m)_{32:63} \neq 0
\]

The contents of the low-order 32 bits of register RX are shifted right the number of bits specified by \((RY)_{58:63}\). Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into RX_{32:63}, RX_{0:31} are set to 0. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

- None
5.11 Move To/From System Register Instructions

The VLE category provides 16-bit forms of instructions to move to/from the LR and CTR.

The fixed-point Move To/From System Register instructions from Book I, *mfspr, mtcfr, mfcfr, mfdcrx, mtocfr, mcrxr, mtmdcx, mtmdcrx, mfdcrux*, and *mtspr* are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.16 of Book I for the instruction definitions.

The fixed-point Move To/From System Register instructions from Book III-E, *mfspr<:E.DC>, mtspr, mfdcr, mtmdcr<:E.DC>, mtmsr, mftmsr, wrtee*, and *wrteei* are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E; see Section 5.4.1 of Book III-E for the instruction definitions.

<table>
<thead>
<tr>
<th>Move From Count Register</th>
<th>R-form</th>
<th>Move From Link Register</th>
<th>R-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>se_mfctr</code> RX</td>
<td></td>
<td><code>se_mflr</code> RX</td>
<td></td>
</tr>
<tr>
<td>0 6 10 RX</td>
<td></td>
<td>0 8 RX</td>
<td></td>
</tr>
</tbody>
</table>

RX ← CTR

The CTR contents are placed into register RX.

Special Registers Altered:
None

Move From Count Register R-form

<table>
<thead>
<tr>
<th>Move To Count Register</th>
<th>R-form</th>
<th>Move To Link Register</th>
<th>R-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>se_mtctr</code> RX</td>
<td></td>
<td><code>se_mtlr</code> RX</td>
<td></td>
</tr>
<tr>
<td>0 6 11 RX</td>
<td></td>
<td>0 9 RX</td>
<td></td>
</tr>
</tbody>
</table>

CTR ← (RX)

The contents of register RX are placed into the CTR.

Special Registers Altered:
CTR

Move To Count Register R-form

<table>
<thead>
<tr>
<th>Move To Link Register</th>
<th>R-form</th>
<th>Move To Link Register</th>
<th>R-form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>se_mtlr</code> RX</td>
<td></td>
<td><code>se_mtlr</code> RX</td>
<td></td>
</tr>
<tr>
<td>0 6 9 RX</td>
<td></td>
<td>0 9 RX</td>
<td></td>
</tr>
</tbody>
</table>

LR ← (RX)

The contents of register RX are placed into the LR.

Special Registers Altered:
LR
Chapter 6. Storage Control Instructions

6.1 Storage Synchronization Instructions

The memory synchronization instructions implemented by category VLE are identical in semantics to those defined in Book II and Book III-E. The se_isync instruction is defined by category VLE, but has the same semantics as isync.

The Load and Reserve and Store Conditional instructions from Book II, lbarx, lharx, lwarx, stbcx, sthcx, and stwcx, are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 4.4.2 of Book II for the instruction definitions.

The Load and Reserve and Store Conditional instructions from Book II, ldarx and stdcx, are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 4.4.2 of Book II for the instruction definitions.

The Memory Barrier instructions from Book II, sync and mbar are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 4.4.3 of Book II for the instruction definitions.

The wait instruction from Book II is available while executing in VLE mode if the category Wait is implemented. The mnemonics, decoding, and semantics for wait are identical to those in Book II; see Section 4.4 of Book II for the instruction definition.

Instruction Synchronize C-form

se_isync

01 15

Executing an se_isync instruction ensures that all instructions preceding the se_isync instruction have completed before the se_isync instruction completes, and that no subsequent instructions are initiated until after the se_isync instruction completes. It also ensures that all instruction cache block invalidations caused by icbi instructions preceding the se_isync instruction have been performed with respect to the processor executing the se_isync instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the se_isync instruction may complete before storage accesses associated with instructions preceding the se_isync instruction have been performed. This instruction is context synchronizing.

The se_isync instruction has identical semantics to the Book II isync instruction, but has a different encoding.

Special Registers Altered:

None
6.2 Cache Management Instructions

Cache management instructions implemented by category VLE are identical to those defined in Book II and Book III-E.

The Cache Management instructions from Book II, \texttt{dcba}, \texttt{dcbf}, \texttt{dcbst}, \texttt{dcbt}, \texttt{dcbtst}, \texttt{dcbz}, \texttt{icbi}, and \texttt{icbt} are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book II; see Section 4.3 of Book II for the instruction definitions.

The Cache Management instruction from Book III-E, \texttt{dcbi} is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction are identical to those in Book III-E; see Section 6.11.1 of Book III-E for the instruction definition.

6.3 Cache Locking Instructions

Cache locking instructions implemented by category VLE are identical to those defined in Book III-E. If the Cache Locking instructions are implemented in category VLE, the Category: Embedded Cache Locking must also be implemented.

The Cache Locking instructions from Book III-E, \texttt{dcbtls}, \texttt{dcbtstls}, \texttt{dcblc}, \texttt{dcblq}, \texttt{icbtls}, \texttt{icblq}, and \texttt{icblc} are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E; see Section 6.11.2 of Book III-E for the instruction definitions.

6.4 TLB Management Instructions

The TLB Management instructions implemented by category VLE are identical to those defined in Book III-E.

The TLB Management instructions from Book III-E, \texttt{tlbre}, \texttt{tlbwe}, \texttt{tlbivax}, \texttt{tlbilx}, \texttt{tlbsync}, \texttt{tlbsrx}, \texttt{<E.TWC>}, and \texttt{tlbsx} are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E. See Section 6.11.4.9 of Book III-E.

Instructions and resources described in Chapter 6 of Book III-E are available if the appropriate category is implemented.

6.5 Instruction Alignment and Byte Ordering

Only Big-Endian instruction memory is supported when executing from a page of VLE instructions. Attempting to fetch VLE instructions from a page marked as Little-Endian generates an instruction storage interrupt byte-ordering exception.
Chapter 7. Additional Categories Available in VLE

Instructions and resources from categories other than Base and Embedded are available in VLE. These include categories for which all the instructions in the category use primary opcode 4 or primary opcode 31.

7.1 Move Assist

Move Assist instructions implemented by category VLE are identical to those defined in Book I. If category Move Assist is supported in non-VLE mode, Move Assist instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.7 of Book I for the instruction definitions.

7.2 Vector

Vector instructions implemented by category VLE are identical to those defined in Book I. If category Vector is supported in non-VLE mode, Vector instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 5 of Book I for the instruction definitions.

7.3 Signal Processing Engine

Signal Processing Engine instructions implemented by category VLE are identical to those defined in Book I. If category Signal Processing Engine is supported in non-VLE mode, Signal Processing Engine instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 8 of Book I for the instruction definitions.

7.4 Embedded Floating Point

Embedded Floating Point instructions implemented by category VLE are identical to those defined in Book I. If category SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, or SPE.Embedded Float Vector is supported in non-VLE mode, the appropriate Embedded Floating Point instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 9 of Book I for the instruction definitions.

7.5 Legacy Move Assist

Legacy Move Assist instructions implemented by category VLE are identical to those defined in Book I. If category Legacy Move Assist is supported in non-VLE mode, Legacy Move Assist instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 10 of Book I for the instruction definitions.

7.6 Embedded Hypervisor

Embedded Hypervisor instructions implemented by category VLE are not identical to those defined in Book III - E. The _ehpriv instruction is identical in mnemonics, decoding, and semantics to the instruction defined in Book III-E. See Section 4.3.1 of Book III-E for the instruction definition. The sc instruction which provides a LEV field for executing calls to the hypervisor software is implemented as e_sc, which is defined in Section 4.3 of Book VLE. The rfgi instruction is implemented as se_rfgi, which is also defined in Section 4.3 of Book VLE. If Category: Embedded Hypervisor is supported in non-VLE mode, Embedded Hypervisor instructions are also supported in VLE mode.

7.7 External PID

External Process ID instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded External PID is supported in non-VLE mode, External Process ID instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Chapter 5.3.7 of Book III-E for the instruction definitions.
7.8 Embedded Performance Monitor

*Embedded Performance Monitor* instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Performance Monitor is supported in non-VLE mode, *Embedded Performance Monitor* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Appendix D of Book III-E for the instruction definitions.

7.9 Processor Control

*Processor Control* instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Processor Control is supported in non-VLE mode, *Processor Control* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Chapter 11 for the instruction definitions.

7.10 Decorated Storage

*Decorated Storage* instructions implemented by category VLE are identical to those defined in Book II. If category Decorated Storage is supported in non-VLE mode, Decorated Storage instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Chapter 8 of Book II for the instruction definitions.

7.11 Embedded Cache Initialization

*Embedded Cache Initialization* instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Cache Initialization is supported in non-VLE mode, *Embedded Cache Initialization* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; Chapter A.1 of Book III-E for the instruction definitions.

7.12 Embedded Cache Debug

*Embedded Cache Debug* instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Cache Debug is supported in non-VLE mode, *Embedded Cache Debug* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; Chapter A.2 of Book III-E for the instruction definitions.
## Appendix A. VLE Instruction Set Sorted by Mnemonic

This appendix lists all the instructions available in VLE mode in the Power ISA, in order by mnemonic. Opcodes that are not defined below are treated as illegal by category VLE.

<table>
<thead>
<tr>
<th>Form</th>
<th>Opcode (hexadecimal)</th>
<th>Mode</th>
<th>Dep.</th>
<th>Priv.</th>
<th>Cat 1</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>XO</td>
<td>7C000214</td>
<td>SR</td>
<td>B</td>
<td>add[.]</td>
<td>7C0000214</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>XO</td>
<td>7C000114</td>
<td>SR</td>
<td>B</td>
<td>addc[.]</td>
<td>7C0000114</td>
<td>Add Carrying</td>
<td>Add Carrying</td>
</tr>
<tr>
<td>XO</td>
<td>7C0001D4</td>
<td>SR</td>
<td>B</td>
<td>addde[.]</td>
<td>7C00001D4</td>
<td>Add Extended</td>
<td>Add Extended</td>
</tr>
<tr>
<td>XO</td>
<td>7C000194</td>
<td>SR</td>
<td>B</td>
<td>adde[.]</td>
<td>7C0000194</td>
<td>Add One Extended</td>
<td>Add One Extended</td>
</tr>
<tr>
<td>X</td>
<td>7C000038</td>
<td>SR</td>
<td>B</td>
<td>and[.]</td>
<td>7C0000038</td>
<td>AND</td>
<td>AND</td>
</tr>
<tr>
<td>X</td>
<td>7C000078</td>
<td>SR</td>
<td>B</td>
<td>andc[.]</td>
<td>7C000078</td>
<td>AND with Complement</td>
<td>AND with Complement</td>
</tr>
<tr>
<td>EVX</td>
<td>1000020F</td>
<td>SP</td>
<td>brinc</td>
<td>Bit Reversed Increment</td>
<td>Bit Reversed Increment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000000</td>
<td>B</td>
<td>cmp</td>
<td>Compare</td>
<td>Compare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000040</td>
<td>B</td>
<td>cmpl</td>
<td>Compare Logical</td>
<td>Compare Logical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000074</td>
<td>SR</td>
<td>64</td>
<td>cntld[.]</td>
<td>7C000074</td>
<td>Count Leading Zeros Doubleword</td>
<td>Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>7C000034</td>
<td>SR</td>
<td>64</td>
<td>cntlw[.]</td>
<td>7C000034</td>
<td>Count Leading Zeros Word</td>
<td>Count Leading Zeros Word</td>
</tr>
<tr>
<td>X</td>
<td>7C0005EC</td>
<td>E</td>
<td>dcba</td>
<td>Data Cache Block Allocate</td>
<td>Data Cache Block Allocate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000AC</td>
<td>B</td>
<td>dcbf</td>
<td>Data Cache Block Flush</td>
<td>Data Cache Block Flush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000FE</td>
<td>E.PD</td>
<td>dbcfep</td>
<td>Data Cache Block Flush by External PID</td>
<td>Data Cache Block Flush by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003AC</td>
<td>E</td>
<td>dbci</td>
<td>Data Cache Block Invalidate</td>
<td>Data Cache Block Invalidate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00030C</td>
<td>M</td>
<td>dbcl</td>
<td>Data Cache Block Lock Clear</td>
<td>Data Cache Block Lock Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00034D</td>
<td>M</td>
<td>dbblq</td>
<td>Data Cache Block Lock Query</td>
<td>Data Cache Block Lock Query</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006C</td>
<td>B</td>
<td>dbst</td>
<td>Data Cache Block Store</td>
<td>Data Cache Block Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00007E</td>
<td>E.PD</td>
<td>dbcbstep</td>
<td>Data Cache Block Store by External PID</td>
<td>Data Cache Block Store by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00022C</td>
<td>B</td>
<td>dcbt</td>
<td>Data Cache Block Touch</td>
<td>Data Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00027E</td>
<td>E.PD</td>
<td>dcbtstep</td>
<td>Data Cache Block Touch by External PID</td>
<td>Data Cache Block Touch by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00014C</td>
<td>ECL</td>
<td>dcbtls</td>
<td>Data Cache Block Touch and Lock Set</td>
<td>Data Cache Block Touch and Lock Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001EC</td>
<td>ECL</td>
<td>dcbtst</td>
<td>Data Cache Block Touch for Store</td>
<td>Data Cache Block Touch for Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001FE</td>
<td>E.PD</td>
<td>dcbtstep</td>
<td>Data Cache Block Touch for External PID</td>
<td>Data Cache Block Touch for External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00010C</td>
<td>M</td>
<td>dcbts</td>
<td>Data Cache Block Touch for Store and Lock Set</td>
<td>Data Cache Block Touch for Store and Lock Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0007EC</td>
<td>B</td>
<td>dcbz</td>
<td>Data Cache Block set to Zero</td>
<td>Data Cache Block set to Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0007FE</td>
<td>E.PD</td>
<td>dcbzep</td>
<td>Data Cache Block set to Zero by External PID</td>
<td>Data Cache Block set to Zero by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00038C</td>
<td>H</td>
<td>dci</td>
<td>Data Cache Invalidate</td>
<td>Data Cache Invalidate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003CC</td>
<td>H</td>
<td>E.CD</td>
<td>dcrit</td>
<td>Data Cache Read [Alternative Encoding]</td>
<td>Data Cache Read [Alternative Encoding]</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C0003D2</td>
<td>SR</td>
<td>64</td>
<td>divd[.]</td>
<td>7C00003D2</td>
<td>Divide Doubleword</td>
<td>Divide Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>7C000392</td>
<td>SR</td>
<td>64</td>
<td>divdu[.]</td>
<td>7C0000392</td>
<td>Divide Doubleword Unsigned</td>
<td>Divide Doubleword Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>7C0003D6</td>
<td>SR</td>
<td>64</td>
<td>divw[.]</td>
<td>7C00003D6</td>
<td>Divide Word</td>
<td>Divide Word</td>
</tr>
<tr>
<td>XO</td>
<td>7C000396</td>
<td>SR</td>
<td>64</td>
<td>divw[.]</td>
<td>7C0000396</td>
<td>Divide Word Unsigned</td>
<td>Divide Word Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>7C00009C</td>
<td>LRV</td>
<td>dlmzb[.]</td>
<td>Determine Leftmost Zero Byte</td>
<td>Determine Leftmost Zero Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003C6</td>
<td>DS</td>
<td>dsn</td>
<td>Decorated Storage Notify</td>
<td>Decorated Storage Notify</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1C000000</td>
<td>VLE</td>
<td>e_add16i</td>
<td>Add Immediate</td>
<td>Add Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>70008000</td>
<td>VLE</td>
<td>e_add2i.</td>
<td>Add (2 operand) Immediate and Record</td>
<td>Add (2 operand) Immediate and Record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>70009000</td>
<td>VLE</td>
<td>e_add2is</td>
<td>Add (2 operand) Immediate Shifted</td>
<td>Add (2 operand) Immediate Shifted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC18</td>
<td>18008000</td>
<td>VLE</td>
<td>e_addi.</td>
<td>Add Scaled Immediate</td>
<td>Add Scaled Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC18</td>
<td>18009000</td>
<td>VLE</td>
<td>e_addic[.]</td>
<td>Add Scaled Immediate Carrying</td>
<td>Add Scaled Immediate Carrying</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000C800</td>
<td>VLE</td>
<td>e_and2i.</td>
<td>AND (two operand) Immediate</td>
<td>AND (two operand) Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000E800</td>
<td>VLE</td>
<td>e_and2is.</td>
<td>AND (2 operand) Immediate Shifted</td>
<td>AND (2 operand) Immediate Shifted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC18</td>
<td>1800C000</td>
<td>VLE</td>
<td>e_and[.]</td>
<td>AND Scaled Immediate</td>
<td>AND Scaled Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BD24</td>
<td>78000000</td>
<td>VLE</td>
<td>e_b[.]</td>
<td>Branch [and Link]</td>
<td>Branch [and Link]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep. Priv.</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>----------------</td>
<td>-------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BD15</td>
<td>7A000000</td>
<td>CT</td>
<td>VLE</td>
<td>e_bc[i]</td>
<td>Branch Conditional [and Link]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>70009800</td>
<td></td>
<td>VLE</td>
<td>e_cmp16i</td>
<td>Compare Immediate Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001C</td>
<td></td>
<td>VLE</td>
<td>e_cmph</td>
<td>Compare Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>70009800</td>
<td></td>
<td>VLE</td>
<td>e_cmph16i</td>
<td>Compare Halfword Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005C</td>
<td></td>
<td>VLE</td>
<td>e_cmphl</td>
<td>Compare Halfword Logical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>7000B000</td>
<td></td>
<td>VLE</td>
<td>e_cmpi</td>
<td>Compare Scaled Immediate Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>1800A800</td>
<td></td>
<td>VLE</td>
<td>e_cmp16i</td>
<td>Compare Logical Immediate Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>1880A800</td>
<td></td>
<td>VLE</td>
<td>e_cmpl</td>
<td>Compare Logical Scaled Immediate Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000102</td>
<td></td>
<td>VLE</td>
<td>e_crand</td>
<td>Condition Register AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C0001C2</td>
<td></td>
<td>VLE</td>
<td>e_crnand</td>
<td>Condition Register NAND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C00042</td>
<td></td>
<td>VLE</td>
<td>e_crech</td>
<td>Condition Register OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000342</td>
<td></td>
<td>VLE</td>
<td>e_crnc</td>
<td>Condition Register OR with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>30000000</td>
<td></td>
<td>VLE</td>
<td>e_lbz</td>
<td>Load Byte and Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000000</td>
<td></td>
<td>VLE</td>
<td>e_lbzu</td>
<td>Load Byte and Zero with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>38000000</td>
<td></td>
<td>VLE</td>
<td>e_lha</td>
<td>Load Halfword Algebraic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000300</td>
<td></td>
<td>VLE</td>
<td>e_lhau</td>
<td>Load Halfword Algebraic with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>58000000</td>
<td></td>
<td>VLE</td>
<td>e_lhz</td>
<td>Load Halfword and Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000100</td>
<td></td>
<td>VLE</td>
<td>e_lhzu</td>
<td>Load Halfword and Zero with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LI20</td>
<td>70000000</td>
<td></td>
<td>VLE</td>
<td>e_li</td>
<td>Load Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000B000</td>
<td></td>
<td>VLE</td>
<td>e_lis</td>
<td>Load Immediate Shifted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000800</td>
<td></td>
<td>VLE</td>
<td>e_lm</td>
<td>Load Multiple Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>50000000</td>
<td></td>
<td>VLE</td>
<td>e_lw</td>
<td>Load Word and Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000200</td>
<td></td>
<td>VLE</td>
<td>e_lwu</td>
<td>Load Word and Zero with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000202</td>
<td></td>
<td>VLE</td>
<td>e_mcrf</td>
<td>Move CR Field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>7000A000</td>
<td></td>
<td>VLE</td>
<td>e_multi2i</td>
<td>Multiply (2 operand) Low Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>1800A000</td>
<td></td>
<td>VLE</td>
<td>e_multiply</td>
<td>Multiply Low Scaled Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000C000</td>
<td></td>
<td>VLE</td>
<td>e_or2i</td>
<td>OR (two operand) Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000D000</td>
<td></td>
<td>VLE</td>
<td>e_or2is</td>
<td>OR (2 operand) Immediate Shifted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>1800D000</td>
<td>SR</td>
<td>VLE</td>
<td>e_or[i]</td>
<td>OR Scaled Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00230</td>
<td>SR</td>
<td>VLE</td>
<td>e_rlwi[i]</td>
<td>Rotate Left Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>74000000</td>
<td>SR</td>
<td>VLE</td>
<td>e_rlwinm</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>74000001</td>
<td></td>
<td>VLE</td>
<td>e_rlwinm</td>
<td>Rotate Left Word Immediate then AND with Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESC</td>
<td>7C000048</td>
<td>E.HV</td>
<td>VLE</td>
<td>e_sc</td>
<td>System Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00070</td>
<td>SR</td>
<td>VLE</td>
<td>e_slwi[i]</td>
<td>Shift Left Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000470</td>
<td>SR</td>
<td>VLE</td>
<td>e_srwi[i]</td>
<td>Shift Right Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>34000000</td>
<td></td>
<td>VLE</td>
<td>e_stb</td>
<td>Store Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000400</td>
<td></td>
<td>VLE</td>
<td>e_stbu</td>
<td>Store Byte with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>5C000000</td>
<td></td>
<td>VLE</td>
<td>e_sth</td>
<td>Store Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000500</td>
<td></td>
<td>VLE</td>
<td>e_sth</td>
<td>Store Halfword with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>18000900</td>
<td></td>
<td>VLE</td>
<td>e_pwm</td>
<td>Store Multiple Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>54000000</td>
<td></td>
<td>VLE</td>
<td>e_stw</td>
<td>Store Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000600</td>
<td></td>
<td>VLE</td>
<td>e_stwu</td>
<td>Store Word with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>1800B000</td>
<td>SR</td>
<td>VLE</td>
<td>e_subf[i]</td>
<td>Subtract From Scaled Immediate Carrying</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>1800C000</td>
<td>SR</td>
<td>VLE</td>
<td>e_xor[i]</td>
<td>XOR Scaled Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E4</td>
<td></td>
<td>SPFD</td>
<td>efdbabs</td>
<td>Floating-Point Double-Precision Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E0</td>
<td></td>
<td>SPFD</td>
<td>efddadd</td>
<td>Floating-Point Double-Precision Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002ED</td>
<td></td>
<td>SPFD</td>
<td>efdfs</td>
<td>Floating-Point Double-Precision Convert from Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F3</td>
<td></td>
<td>SPFD</td>
<td>efdfsfs</td>
<td>Convert Floating-Point Double-Precision from Signed Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F1</td>
<td></td>
<td>SPFD</td>
<td>efdfsfi</td>
<td>Convert Floating-Point Double-Precision from Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E3</td>
<td></td>
<td>SPFD</td>
<td>efdfsids</td>
<td>Convert Floating-Point Double-Precision from Signed Integer Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode</td>
<td>Dep^1 Priv</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F2</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdcfuf</td>
<td>Convert Floating-Point Double-Precision from Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F0</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdcfui</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E2</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdcfuid</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer Doubleword</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002EE</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdcmpeq</td>
<td>Floating-Point Double-Precision Compare Equal</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002EC</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdcmpgt</td>
<td>Floating-Point Double-Precision Compare Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002ED</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdcmplt</td>
<td>Floating-Point Double-Precision Compare Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F7</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctsf</td>
<td>Convert Floating-Point Double-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F5</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctsi</td>
<td>Convert Floating-Point Double-Precision to Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002EB</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctsidz</td>
<td>Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FA</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctsz</td>
<td>Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F6</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctuf</td>
<td>Convert Floating-Point Double-Precision to Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F4</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctui</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002EA</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctuidz</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F8</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdctuiz</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E9</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efddiv</td>
<td>Floating-Point Double-Precision Divide</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E8</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdmul</td>
<td>Floating-Point Double-Precision Multiply</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E5</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdnabs</td>
<td>Floating-Point Double-Precision Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E6</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdneg</td>
<td>Floating-Point Double-Precision Negate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002E1</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdsub</td>
<td>Floating-Point Double-Precision Subtract</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FE</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdsteg</td>
<td>Floating-Point Double-Precision Test Equal</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FC</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdstgt</td>
<td>Floating-Point Double-Precision Test Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FD</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efdstlt</td>
<td>Floating-Point Double-Precision Test Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C4</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsabs</td>
<td>Floating-Point Single-Precision Absolute Value</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C0</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsadd</td>
<td>Floating-Point Single-Precision Add</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002CF</td>
<td>SP:FD</td>
<td></td>
<td></td>
<td>efsclfd</td>
<td>Floating-Point Single-Precision Convert from Double-Precision</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D3</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclfsf</td>
<td>Convert Floating-Point Single-Precision from Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D1</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclfsi</td>
<td>Convert Floating-Point Single-Precision from Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D2</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclfuf</td>
<td>Convert Floating-Point Single-Precision from Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D0</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclfu</td>
<td>Convert Floating-Point Single-Precision from Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002CE</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efscmpeq</td>
<td>Floating-Point Single-Precision Compare Equal</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002CC</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efscmpgt</td>
<td>Floating-Point Single-Precision Compare Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002CD</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efscmplt</td>
<td>Floating-Point Single-Precision Compare Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D7</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclfsf</td>
<td>Convert Floating-Point Single-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D5</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efscltsi</td>
<td>Convert Floating-Point Single-Precision to Signed Integer with Round towards Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002DA</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efscltsiz</td>
<td>Convert Floating-Point Single-Precision to Signed Integer with Round towards Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D6</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclfuf</td>
<td>Convert Floating-Point Single-Precision to Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D4</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsclui</td>
<td>Convert Floating-Point Single-Precision to Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002D8</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efscluiz</td>
<td>Convert Floating-Point Single-Precision to Unsigned Integer with Round towards Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C9</td>
<td>SP:FS</td>
<td></td>
<td></td>
<td>efsdiv</td>
<td>Floating-Point Single-Precision Divide</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep. Priv.</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C8</td>
<td>SP.FS</td>
<td></td>
<td>efsmul</td>
<td>Floating-Point Single-Precision Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C5</td>
<td>SP.FS</td>
<td></td>
<td>efsnabs</td>
<td>Floating-Point Single-Precision Negative Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C6</td>
<td>SP.FS</td>
<td></td>
<td>efsneg</td>
<td>Floating-Point Single-Precision Negate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C1</td>
<td>SP.FS</td>
<td></td>
<td>efssub</td>
<td>Floating-Point Single-Precision Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002DE</td>
<td>SP.FS</td>
<td></td>
<td>efstseq</td>
<td>Floating-Point Single-Precision Test Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002DC</td>
<td>SP.FS</td>
<td></td>
<td>efstgt</td>
<td>Floating-Point Single-Precision Test Greater Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002DD</td>
<td>SP.FS</td>
<td></td>
<td>efstslt</td>
<td>Floating-Point Single-Precision Test Less Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C00021C</td>
<td>E.HV</td>
<td></td>
<td>ehpriv</td>
<td>Embedded Hypervisor Privilege</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000238</td>
<td>B</td>
<td></td>
<td>eqv[,]</td>
<td>Equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000208</td>
<td>SP evabs</td>
<td></td>
<td>evabs</td>
<td>Vector Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000202</td>
<td>SP evaddw</td>
<td></td>
<td>evaddw</td>
<td>Vector Add Immediate Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C9</td>
<td>SP evaddsmiaaw</td>
<td></td>
<td>evaddsmiaaw</td>
<td>Vector Add Signed, Modulo, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C1</td>
<td>SP evaddssiaaw</td>
<td></td>
<td>evaddssiaaw</td>
<td>Vector Add Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C8</td>
<td>SP evaddumiaaw</td>
<td></td>
<td>evaddumiaaw</td>
<td>Vector Add Unsigned, Modulo, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004CD</td>
<td>SP evaddusiaaw</td>
<td></td>
<td>evaddusiaaw</td>
<td>Vector Add Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000200</td>
<td>SP evaddw</td>
<td></td>
<td>evaddw</td>
<td>Vector Add Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000211</td>
<td>SP evand</td>
<td></td>
<td>evand</td>
<td>Vector AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000212</td>
<td>SP evandc</td>
<td></td>
<td>evandc</td>
<td>Vector AND with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000234</td>
<td>SP evcmpeq</td>
<td></td>
<td>evcmpeq</td>
<td>Vector Compare Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000231</td>
<td>SP evcmpgtsl</td>
<td></td>
<td>evcmpgtsl</td>
<td>Vector Compare Greater Than Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000230</td>
<td>SP evcmpgtu</td>
<td></td>
<td>evcmpgtu</td>
<td>Vector Compare Greater Than Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000233</td>
<td>SP evcmpltl</td>
<td></td>
<td>evcmpltl</td>
<td>Vector Compare Less Than Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000232</td>
<td>SP evcmpltu</td>
<td></td>
<td>evcmpltu</td>
<td>Vector Compare Less Than Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000020E</td>
<td>SP evcntlsw</td>
<td></td>
<td>evcntlsw</td>
<td>Vector Count Leading Signed Bits Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000020D</td>
<td>SP evcntlzr</td>
<td></td>
<td>evcntlzr</td>
<td>Vector Count Leading Zeros Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C6</td>
<td>SP evdivws</td>
<td></td>
<td>evdivws</td>
<td>Vector Divide Word Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C7</td>
<td>SP evdivwu</td>
<td></td>
<td>evdivwu</td>
<td>Vector Divide Word Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000219</td>
<td>SP evequiv</td>
<td></td>
<td>evequiv</td>
<td>Vector Equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000020A</td>
<td>SP evextsb</td>
<td></td>
<td>evextsb</td>
<td>Vector Extend Sign Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000020B</td>
<td>SP evextsh</td>
<td></td>
<td>evextsh</td>
<td>Vector Extend Sign Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000284</td>
<td>SP.FV evfsabs</td>
<td></td>
<td>evfsabs</td>
<td>Vector Floating-Point Single-Precision Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000280</td>
<td>SP.FV evfsadd</td>
<td></td>
<td>evfsadd</td>
<td>Vector Floating-Point Single-Precision Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000293</td>
<td>SP.FV evfsclfsf</td>
<td></td>
<td>evfsclfsf</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000291</td>
<td>SP.FV evfsclfsi</td>
<td></td>
<td>evfsclfsi</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000292</td>
<td>SP.FV evfsclfuf</td>
<td></td>
<td>evfsclfuf</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000290</td>
<td>SP.FV evfsclfui</td>
<td></td>
<td>evfsclfui</td>
<td>Vector Convert Floating-Point Single-Precision from Unsigned Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000028E</td>
<td>SP.FV evfscmpeq</td>
<td></td>
<td>evfscmpeq</td>
<td>Vector Floating-Point Single-Precision Compare Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000028C</td>
<td>SP.FV evfscmpgt</td>
<td></td>
<td>evfscmpgt</td>
<td>Vector Floating-Point Single-Precision Compare Greater Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000028D</td>
<td>SP.FV evfsclmplt</td>
<td></td>
<td>evfsclmplt</td>
<td>Vector Floating-Point Single-Precision Compare Less Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000297</td>
<td>SP.FV evfsclmorf</td>
<td></td>
<td>evfsclmorf</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000295</td>
<td>SP.FV evfscltsf</td>
<td></td>
<td>evfscltsf</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000029A</td>
<td>SP.FV evfscltsi</td>
<td></td>
<td>evfscltsi</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000296</td>
<td>SP.FV evfscltsiz</td>
<td></td>
<td>evfscltsiz</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000294</td>
<td>SP.FV evfscltui</td>
<td></td>
<td>evfscltui</td>
<td>Vector Convert Floating-Point Single-Precision to Unsigned Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000298</td>
<td>SP.FV evfscltuiz</td>
<td></td>
<td>evfscltuiz</td>
<td>Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode (hexadecimal)²</td>
<td>Mode¹</td>
<td>Dep.¹</td>
<td>Priv.¹</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>EVX 10000289</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evlsdiv</td>
<td>Vector Floating-Point Single-Precision Divide</td>
<td></td>
</tr>
<tr>
<td>EVX 10000288</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfsmul</td>
<td>Vector Floating-Point Single-Precision Multiply</td>
<td></td>
</tr>
<tr>
<td>EVX 10000285</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfsnabs</td>
<td>Vector Floating-Point Single-Precision Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>EVX 10000286</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfseneg</td>
<td>Vector Floating-Point Single-Precision Negate</td>
<td></td>
</tr>
<tr>
<td>EVX 10000281</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfssub</td>
<td>Vector Floating-Point Single-Precision Subtract</td>
<td></td>
</tr>
<tr>
<td>EVX 1000029E</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfststeq</td>
<td>Vector Floating-Point Single-Precision Test Equal</td>
<td></td>
</tr>
<tr>
<td>EVX 1000029C</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfststgt</td>
<td>Vector Floating-Point Single-Precision Test Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX 1000029D</td>
<td></td>
<td></td>
<td></td>
<td>SF.FV</td>
<td>evfststlt</td>
<td>Vector Floating-Point Single-Precision Test Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX 10000301</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evidd</td>
<td>Vector Load Double Word into Double Word</td>
<td></td>
</tr>
<tr>
<td>EVX 7C00063E</td>
<td>P</td>
<td></td>
<td></td>
<td>E.PD</td>
<td>eviddpx</td>
<td>Vector Load Doubleword into Doubleword by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000300</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlddx</td>
<td>Vector Load Double Word into Double Word Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000305</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evldh</td>
<td>Vector Load Double Word into Four Halfwords</td>
<td></td>
</tr>
<tr>
<td>EVX 10000304</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evldhx</td>
<td>Vector Load Double Word into Four Halfwords Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000303</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evldw</td>
<td>Vector Load Double Word into Two Words</td>
<td></td>
</tr>
<tr>
<td>EVX 10000302</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evldwx</td>
<td>Vector Load Double Word into Two Words Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000309</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlhhesplaat</td>
<td>Vector Load Halfword into Halfwords Even and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX 10000308</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlhesplaatx</td>
<td>Vector Load Halfword into Halfwords Even and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 1000030F</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlhhosplaat</td>
<td>Vector Load Halfword into Halfword Odd and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX 1000030E</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlhhosplaatx</td>
<td>Vector Load Halfword into Halfword Odd Signed and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 1000030D</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlhousingplaat</td>
<td>Vector Load Halfword into Halfword Odd Unsigned and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX 1000030C</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlhousingplaatx</td>
<td>Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000311</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhae</td>
<td>Vector Load Word into Two Halfwords Even</td>
<td></td>
</tr>
<tr>
<td>EVX 10000310</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhex</td>
<td>Vector Load Word into Two Halfwords Even Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000317</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhos</td>
<td>Vector Load Word into Two Halfwords Odd Signed (with sign extension)</td>
<td></td>
</tr>
<tr>
<td>EVX 10000316</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhosx</td>
<td>Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)</td>
<td></td>
</tr>
<tr>
<td>EVX 10000315</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhou</td>
<td>Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)</td>
<td></td>
</tr>
<tr>
<td>EVX 10000314</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhoux</td>
<td>Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)</td>
<td></td>
</tr>
<tr>
<td>EVX 1000031D</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhsplat</td>
<td>Vector Load Word into Two Halfwords and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX 1000031C</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwhsplax</td>
<td>Vector Load Word into Two Halfwords and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 10000319</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwwsplat</td>
<td>Vector Load Word into Word and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX 10000318</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evlwwsplatx</td>
<td>Vector Load Word into Word and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 1000022C</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmergehi</td>
<td>Vector Merge High</td>
<td></td>
</tr>
<tr>
<td>EVX 1000022E</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmergehilo</td>
<td>Vector Merge High/Low</td>
<td></td>
</tr>
<tr>
<td>EVX 1000022D</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmergelohi</td>
<td>Vector Merge Low/Low</td>
<td></td>
</tr>
<tr>
<td>EVX 1000022F</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhgsmfaa</td>
<td>Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 100005AB</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhgsmfan</td>
<td>Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 10000529</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhgsmiaa</td>
<td>Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 100005A9</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhgmsian</td>
<td>Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 10000528</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhugumia</td>
<td>Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 100005A8</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhugumin</td>
<td>Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 1000040B</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>evmhesmf</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep. Priv.</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000042B</td>
<td>SP</td>
<td>evmhesmla</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000050B</td>
<td>SP</td>
<td>evmhesmfaw</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000058B</td>
<td>SP</td>
<td>evmhesmfanw</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000409</td>
<td>SP</td>
<td>evmhesmi</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000429</td>
<td>SP</td>
<td>evmhesmia</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000509</td>
<td>SP</td>
<td>evmhesmiaaw</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000589</td>
<td>SP</td>
<td>evmhesmianw</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000403</td>
<td>SP</td>
<td>evmheessf</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000423</td>
<td>SP</td>
<td>evmheessfa</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000503</td>
<td>SP</td>
<td>evmheessfaaw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000583</td>
<td>SP</td>
<td>evmheessfanw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000501</td>
<td>SP</td>
<td>evmheessiaaw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000589</td>
<td>SP</td>
<td>evmheessianw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000408</td>
<td>SP</td>
<td>evmheumi</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000428</td>
<td>SP</td>
<td>evmheumia</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000508</td>
<td>SP</td>
<td>evmheumiaaw</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000588</td>
<td>SP</td>
<td>evmheumianw</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000500</td>
<td>SP</td>
<td>evmheusiaaw</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000580</td>
<td>SP</td>
<td>evmheusianw</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000052F</td>
<td>SP</td>
<td>evmhogrsmfaa</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005AF</td>
<td>SP</td>
<td>evmhogrsmfan</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000052D</td>
<td>SP</td>
<td>evmhogrsmiaa</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer, and Accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005AD</td>
<td>SP</td>
<td>evmhogrsmian</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000052C</td>
<td>SP</td>
<td>evmhogrsumiaa</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer, and Accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005AC</td>
<td>SP</td>
<td>evmhogrsumian</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000040F</td>
<td>SP</td>
<td>evmhosmfa</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000042F</td>
<td>SP</td>
<td>evmhosmfaaw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000050F</td>
<td>SP</td>
<td>evmhosmfanw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000058F</td>
<td>SP</td>
<td>evmhosmfnw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000040D</td>
<td>SP</td>
<td>evmhosmi</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1320  Power ISA™ - Book VLE
<table>
<thead>
<tr>
<th>Form</th>
<th>Opcode (hexadecimal)</th>
<th>Mode Dep.</th>
<th>Priv.</th>
<th>Cat¹</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVX</td>
<td>1000042D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhosmia</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhosmiaaw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>1000058D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhosmianw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000407</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossf</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>10000427</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossfa</td>
<td>Vector Multiply Halfwords, Odd, Signed, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>10000507</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossfaaw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000587</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossfianw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000505</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossiaaw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000585</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossianw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>1000040C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhoumi</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>1000042C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhoumia</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhoumiaaw</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>1000058C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhoumianw</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000504</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhousiaaw</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000584</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhousianw</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>100004C4</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmra</td>
<td>Initialize Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>1000044F</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhsmf</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>1000046F</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhsmfa</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>1000044D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhsmi</td>
<td>Vector Multiply Word High Signed, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>1000046D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhsmia</td>
<td>Vector Multiply Word High Signed, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>10000447</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhssf</td>
<td>Vector Multiply Word High Signed, Saturate, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>10000467</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhssfa</td>
<td>Vector Multiply Word High Signed, Saturate, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>1000044C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhumi</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>1000046C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwhumia</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>10000549</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlsmiaaw</td>
<td>Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>100005C9</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlsmianw</td>
<td>Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000541</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlsiaaw</td>
<td>Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>100005C1</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlsianw</td>
<td>Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words</td>
</tr>
<tr>
<td>EVX</td>
<td>10000448</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlumi</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>10000468</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlimia</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>10000548</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlimiaaw</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>100005C8</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlimianw</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep. Priv</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000540</td>
<td>SP</td>
<td>evmwlusiaaw</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005C0</td>
<td>SP</td>
<td>evmwlusianw</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000045B</td>
<td>SP</td>
<td>evmwsmf</td>
<td>Vector Multiply Word Signed, Modulo, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000047B</td>
<td>SP</td>
<td>evmwsmfma</td>
<td>Vector Multiply Word Signed, Modulo, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000055B</td>
<td>SP</td>
<td>evmwsmfaaa</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005D0</td>
<td>SP</td>
<td>evmwsmfan</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000459</td>
<td>SP</td>
<td>evmwsmi</td>
<td>Vector Multiply Word Signed, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000479</td>
<td>SP</td>
<td>evmwsmia</td>
<td>Vector Multiply Word Signed, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000559</td>
<td>SP</td>
<td>evmwsmiaaa</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005D9</td>
<td>SP</td>
<td>evmwsmian</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000453</td>
<td>SP</td>
<td>evmwssf</td>
<td>Vector Multiply Word Signed, Saturate, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000473</td>
<td>SP</td>
<td>evmwssfa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000553</td>
<td>SP</td>
<td>evmwssfaaa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005D3</td>
<td>SP</td>
<td>evmwssfan</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000458</td>
<td>SP</td>
<td>evmwumii</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000478</td>
<td>SP</td>
<td>evmwumia</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000558</td>
<td>SP</td>
<td>evmwumiaaa</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer and Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100005D8</td>
<td>SP</td>
<td>evmwumian</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000021E</td>
<td>SP</td>
<td>evnand</td>
<td>Vector NAND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000209</td>
<td>SP</td>
<td>evneg</td>
<td>Vector Negate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000218</td>
<td>SP</td>
<td>evnor</td>
<td>Vector NOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000217</td>
<td>SP</td>
<td>evor</td>
<td>Vector OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000021B</td>
<td>SP</td>
<td>evorc</td>
<td>Vector OR with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000228</td>
<td>SP</td>
<td>evrlw</td>
<td>Vector Rotate Left Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000022A</td>
<td>SP</td>
<td>evrlwi</td>
<td>Vector Rotate Left Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000020C</td>
<td>SP</td>
<td>evrndw</td>
<td>Vector Round Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVS</td>
<td>10000278</td>
<td>SP</td>
<td>evsel</td>
<td>Vector Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000224</td>
<td>SP</td>
<td>evslw</td>
<td>Vector Shift Left Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000226</td>
<td>SP</td>
<td>evslwi</td>
<td>Vector Shift Left Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000022B</td>
<td>SP</td>
<td>evsplati</td>
<td>Vector Splat Fractional Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000229</td>
<td>SP</td>
<td>evsplati</td>
<td>Vector Splat Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000223</td>
<td>SP</td>
<td>evsrwis</td>
<td>Vector Shift Right Word Immediate Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000222</td>
<td>SP</td>
<td>evsrwiu</td>
<td>Vector Shift Right Word Immediate Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000221</td>
<td>SP</td>
<td>evsrws</td>
<td>Vector Shift Right Word Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000220</td>
<td>SP</td>
<td>evsrwu</td>
<td>Vector Shift Right Word Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000321</td>
<td>SP</td>
<td>evstdd</td>
<td>Vector Store Double of Double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>7C00073E</td>
<td>P</td>
<td>evstdddep</td>
<td>Vector Store Doubleword into Doubleword by External Process ID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000320</td>
<td>SP</td>
<td>evstdddx</td>
<td>Vector Store Doubleword of Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000325</td>
<td>SP</td>
<td>evstddh</td>
<td>Vector Store Double of Four Halfwords</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000324</td>
<td>SP</td>
<td>evstdhx</td>
<td>Vector Store Double of Four Halfwords Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000323</td>
<td>SP</td>
<td>evstdw</td>
<td>Vector Store Double of Two Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000322</td>
<td>SP</td>
<td>evstdxw</td>
<td>Vector Store Double of Two Words Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000331</td>
<td>SP</td>
<td>evstwh</td>
<td>Vector Store Word of Two Halfwords from Even</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000330</td>
<td>SP</td>
<td>evstwhex</td>
<td>Vector Store Word of Two Halfwords from Even Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000335</td>
<td>SP</td>
<td>evstwho</td>
<td>Vector Store Word of Two Halfwords from Odd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep. Priv.</td>
<td>Cat</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>----------------</td>
<td>-----</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000334</td>
<td>SP</td>
<td></td>
<td>evstwhox</td>
<td>Vector Store Word of Two Halfwords from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000339</td>
<td>SP</td>
<td></td>
<td>evstwwe</td>
<td>Vector Store Word of Word from Even Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000338</td>
<td>SP</td>
<td></td>
<td>evstwwex</td>
<td>Vector Store Word of Word from Even Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000333</td>
<td>SP</td>
<td></td>
<td>evstwwo</td>
<td>Vector Store Word of Word from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000033C</td>
<td>SP</td>
<td></td>
<td>evstwwox</td>
<td>Vector Store Word of Word from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000004CB</td>
<td>SP</td>
<td></td>
<td>evsubfsmiaaw</td>
<td>Vector Subtract Signed, Module, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000004C3</td>
<td>SP</td>
<td></td>
<td>evsubffsiaaw</td>
<td>Vector Subtract Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000004CA</td>
<td>SP</td>
<td></td>
<td>evsubfumiaaw</td>
<td>Vector Subtract Unsigned, Module, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000004C2</td>
<td>SP</td>
<td></td>
<td>evsubfusiaaw</td>
<td>Vector Subtract Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000204</td>
<td>SP</td>
<td></td>
<td>evsubfw</td>
<td>Vector Subtract from Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000206</td>
<td>SP</td>
<td></td>
<td>evsubifw</td>
<td>Vector Subtract Immediate from Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100000216</td>
<td>SP</td>
<td></td>
<td>evxor</td>
<td>Vector XOR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000774</td>
<td>SR</td>
<td>B</td>
<td>extsb[,]</td>
<td>Extend Sign Byte</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000734</td>
<td>SR</td>
<td>B</td>
<td>extsh[,]</td>
<td>Extend Sign Halfword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00007B4</td>
<td>SR</td>
<td>64</td>
<td>extsw[,]</td>
<td>Extend Sign Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00007AC</td>
<td>B</td>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00007BE</td>
<td>P</td>
<td>E.PD</td>
<td>icbiep</td>
<td>Instruction Cache Block Invalidate by External PID</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001CC</td>
<td>M</td>
<td>ECL</td>
<td>icblc</td>
<td>Instruction Cache Block Lock Clear</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001BD</td>
<td>M</td>
<td>ECL</td>
<td>icbq</td>
<td>Instruction Cache Block Lock Query</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002C</td>
<td>B</td>
<td>icbt</td>
<td>Instruction Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003CC</td>
<td>M</td>
<td>ECL</td>
<td>icbts</td>
<td>Instruction Cache Block Touch and Lock Set</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00078C</td>
<td>H</td>
<td>E.Cl</td>
<td>ici</td>
<td>Instruction Cache Invalidate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000077C</td>
<td>H</td>
<td>E.CD</td>
<td>icread</td>
<td>Instruction Cache Read</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00001E</td>
<td>B</td>
<td>isel</td>
<td>Integer Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000068</td>
<td>B</td>
<td>lbarx</td>
<td>Load Byte and Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000406</td>
<td>DS</td>
<td>ld6x</td>
<td>Load Byte with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00008E</td>
<td>P</td>
<td>E.PD</td>
<td>lbepx</td>
<td>Load Byte by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000EE</td>
<td>B</td>
<td>lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000AE</td>
<td>B</td>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000A8</td>
<td>64</td>
<td>ldax</td>
<td>Load Doubleword And Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0004C6</td>
<td>DS</td>
<td>ld6x</td>
<td>Load Doubleword with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00003A</td>
<td>P</td>
<td>E.PD</td>
<td>ld6px</td>
<td>Load Doubleword by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006A</td>
<td>64</td>
<td>ld6x</td>
<td>Load Doubleword with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002A</td>
<td>64</td>
<td>ld6x</td>
<td>Load Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000046</td>
<td>DS</td>
<td>ld6dx</td>
<td>Load Doubleword with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00004BE</td>
<td>P</td>
<td>E.PD</td>
<td>lfd6px</td>
<td>Load Floating-Point Double by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000E8</td>
<td>B</td>
<td>lhaxx</td>
<td>Load Halfword and Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002EE</td>
<td>B</td>
<td>lhaxx</td>
<td>Load Halfword Algebraic with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002AE</td>
<td>B</td>
<td>lhax</td>
<td>Load Halfword Algebraic Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000062C</td>
<td>B</td>
<td>lhbx</td>
<td>Load Halfword Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000446</td>
<td>DS</td>
<td>lh6dx</td>
<td>Load Halfword with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000023E</td>
<td>P</td>
<td>E.PD</td>
<td>lh6px</td>
<td>Load Halfword by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000026E</td>
<td>B</td>
<td>lh6ux</td>
<td>Load Halfword and Zero with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000022E</td>
<td>B</td>
<td>lh6x</td>
<td>Load Halfword and Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00004AA</td>
<td>MA</td>
<td>lswi</td>
<td>Load String Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000042A</td>
<td>MA</td>
<td>lswx</td>
<td>Load String Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000000E</td>
<td>V</td>
<td>lvebx</td>
<td>Load Vector Element Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000004E</td>
<td>V</td>
<td>lvehx</td>
<td>Load Vector Element Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002AE</td>
<td>P</td>
<td>E.PD</td>
<td>lvepx</td>
<td>Load Vector by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000020E</td>
<td>P</td>
<td>E.PD</td>
<td>lvpxl</td>
<td>Load Vector by External Process ID Indexed LRU</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000008E</td>
<td>V</td>
<td>lvexx</td>
<td>Load Vector Element Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000003C</td>
<td>V</td>
<td>lvsl</td>
<td>Load Vector for Shift Left Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000004C</td>
<td>V</td>
<td>lvsr</td>
<td>Load Vector for Shift Right Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000CE</td>
<td>V</td>
<td>lvx</td>
<td>Load Vector Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002CE</td>
<td>V</td>
<td>lvxl</td>
<td>Load Vector Indexed LRU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000228</td>
<td>B</td>
<td>lwarx</td>
<td>Load Word And Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep.</td>
<td>Priv</td>
<td>Cat</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>-----------</td>
<td>------</td>
<td>-----</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>X</td>
<td>7C0002EA</td>
<td></td>
<td></td>
<td>64</td>
<td>lwaux</td>
<td>Load Word Algebraic with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C0002AA</td>
<td></td>
<td></td>
<td>64</td>
<td>lwax</td>
<td>Load Word Algebraic Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00042C</td>
<td>P</td>
<td></td>
<td></td>
<td>lwbrx</td>
<td>Load Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000486</td>
<td>DS</td>
<td></td>
<td></td>
<td>lwdx</td>
<td>Load Word with Decoration Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00003E</td>
<td>E.PD</td>
<td></td>
<td></td>
<td>lwepx</td>
<td>Load Word by External Process ID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00006E</td>
<td>B</td>
<td></td>
<td></td>
<td>lwzux</td>
<td>Load Word and Zero with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>B</td>
<td></td>
<td></td>
<td>lwzx</td>
<td>Load Word and Zero Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>10000158</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Cross Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>100001D8</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>10000198</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>10000118</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate High Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>10000058</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate High Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>100000D8</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate High Halfword to Word Saturate Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>10000098</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>10000358</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>100003D8</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>10000398</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>10000318</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>10000318</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>7C0006AC</td>
<td>E</td>
<td></td>
<td></td>
<td>mbar</td>
<td>Memory Barrier</td>
</tr>
<tr>
<td>X</td>
<td>7C000000</td>
<td>E</td>
<td></td>
<td></td>
<td>mcrxr</td>
<td>Move To Condition Register from XER</td>
</tr>
<tr>
<td>XFX</td>
<td>7C000026</td>
<td>B</td>
<td></td>
<td></td>
<td>mfcr</td>
<td>Move From Condition Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C000286</td>
<td>P</td>
<td>E.DC</td>
<td></td>
<td>mfdcr</td>
<td>Move From Device Control Register</td>
</tr>
<tr>
<td>X</td>
<td>7C000246</td>
<td>E.DC</td>
<td></td>
<td></td>
<td>mfdcrxx</td>
<td>Move From Device Control Register User-mode Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000206</td>
<td>P</td>
<td>E.DC</td>
<td></td>
<td>mfdcrx</td>
<td>Move From Device Control Register Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C0000A6</td>
<td>P</td>
<td>B</td>
<td></td>
<td>mfmsr</td>
<td>Move From Machine State Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C100100</td>
<td>B</td>
<td></td>
<td></td>
<td>mfocrf</td>
<td>Move From One Condition Register Field</td>
</tr>
<tr>
<td>XFX</td>
<td>7C00029C</td>
<td>O</td>
<td>E.PM</td>
<td></td>
<td>mfpmr</td>
<td>Move From Performance Monitor Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C0002A6</td>
<td>O</td>
<td>B</td>
<td></td>
<td>mfspr</td>
<td>Move From Special Purpose Register</td>
</tr>
<tr>
<td>VX</td>
<td>10000604</td>
<td>V</td>
<td></td>
<td></td>
<td>mvfscr</td>
<td>Move From VSCR</td>
</tr>
<tr>
<td>X</td>
<td>7C0001DC</td>
<td>H</td>
<td>E.PC</td>
<td></td>
<td>msgclr</td>
<td>Message Clear</td>
</tr>
<tr>
<td>X</td>
<td>7C00019C</td>
<td>H</td>
<td>E.PC</td>
<td></td>
<td>msgsnd</td>
<td>Message Send</td>
</tr>
<tr>
<td>XFX</td>
<td>7C000120</td>
<td>B</td>
<td></td>
<td></td>
<td>mtcfr</td>
<td>Move To Condition Register Fields</td>
</tr>
<tr>
<td>XFX</td>
<td>7C000386</td>
<td>P</td>
<td>E.DC</td>
<td></td>
<td>mtocr</td>
<td>Move To Device Control Register</td>
</tr>
<tr>
<td>X</td>
<td>7C000346</td>
<td>E.DC</td>
<td></td>
<td></td>
<td>mtocrxx</td>
<td>Move To Device Control Register User-mode Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000306</td>
<td>P</td>
<td>E.DC</td>
<td></td>
<td>mtocrx</td>
<td>Move To Device Control Register Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000124</td>
<td>P</td>
<td>E</td>
<td></td>
<td>mtnmsr</td>
<td>Move To Machine State Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C100120</td>
<td>B</td>
<td></td>
<td></td>
<td>mtocrf</td>
<td>Move To One Condition Register Field</td>
</tr>
<tr>
<td>XFX</td>
<td>7C00039C</td>
<td>O</td>
<td>E.PM</td>
<td></td>
<td>mtmpr</td>
<td>Move To Performance Monitor Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C0003A6</td>
<td>O</td>
<td>B</td>
<td></td>
<td>mtpsr</td>
<td>Move To Special Purpose Register</td>
</tr>
<tr>
<td>VX</td>
<td>10000644</td>
<td>V</td>
<td></td>
<td></td>
<td>mtvscr</td>
<td>Move To VSCR</td>
</tr>
<tr>
<td>X</td>
<td>10000150</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Cross Halfword to Word Signed</td>
</tr>
<tr>
<td>X</td>
<td>10000110</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply Cross Halfword to Word Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>7C000092</td>
<td>SR</td>
<td></td>
<td></td>
<td>mulhdu</td>
<td>Multiply High Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>7C000012</td>
<td>SR</td>
<td></td>
<td>64</td>
<td>mulhdu</td>
<td>Multiply High Doubleword Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>10000050</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply High Halfword to Word Signed</td>
</tr>
<tr>
<td>X</td>
<td>10000010</td>
<td>SR</td>
<td></td>
<td></td>
<td>LMA</td>
<td>Multiply High Halfword to Word Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>7C000096</td>
<td>SR</td>
<td></td>
<td></td>
<td>multw</td>
<td>Multiply High Word</td>
</tr>
<tr>
<td>XO</td>
<td>7C000016</td>
<td>SR</td>
<td></td>
<td></td>
<td>multw</td>
<td>Multiply High Word Unsigned</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode</td>
<td>Dep.</td>
<td>Cat1</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>XO</td>
<td>7C0001D2</td>
<td>SR</td>
<td>64</td>
<td></td>
<td>mulld[]</td>
<td>Multiply Low Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>10000350</td>
<td>SR</td>
<td></td>
<td></td>
<td>mullhw[]</td>
<td>Multiply Low Halfword to Word Signed</td>
</tr>
<tr>
<td>X</td>
<td>10000310</td>
<td>SR</td>
<td></td>
<td></td>
<td>mullhwu[]</td>
<td>Multiply Low Halfword to Word Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>7C0001D6</td>
<td>B</td>
<td></td>
<td></td>
<td>mulw[]</td>
<td>Multiply Low Word</td>
</tr>
<tr>
<td>X</td>
<td>1000035B</td>
<td>B</td>
<td>nand[.]</td>
<td></td>
<td>NAND</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>100000D2</td>
<td>LMA</td>
<td></td>
<td></td>
<td>nmachw[.]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word</td>
</tr>
<tr>
<td>XO</td>
<td>100000D6</td>
<td>LMA</td>
<td></td>
<td></td>
<td>nmachws[.]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word</td>
</tr>
<tr>
<td>XO</td>
<td>100000F4</td>
<td></td>
<td></td>
<td></td>
<td>nmacch[]</td>
<td>Negative Multiply Accumulate High Halfword to Word</td>
</tr>
<tr>
<td>XO</td>
<td>10000310</td>
<td>LMA</td>
<td></td>
<td></td>
<td>nmachhw[.]</td>
<td>Negative Multiply Accumulate High Halfword to Word</td>
</tr>
<tr>
<td>XO</td>
<td>1000035C</td>
<td>LMA</td>
<td></td>
<td></td>
<td>nmachlw[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word</td>
</tr>
<tr>
<td>XO</td>
<td>100003DC</td>
<td>LMA</td>
<td></td>
<td></td>
<td>nmachhs[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word</td>
</tr>
<tr>
<td>X</td>
<td>7C0000F8</td>
<td>B</td>
<td></td>
<td></td>
<td>nor[.]</td>
<td>NOR</td>
</tr>
<tr>
<td>X</td>
<td>7C00007B</td>
<td>B</td>
<td></td>
<td></td>
<td>or[.]</td>
<td>OR</td>
</tr>
<tr>
<td>X</td>
<td>7C00003B</td>
<td>B</td>
<td></td>
<td></td>
<td>orc[.]</td>
<td>OR with Complement</td>
</tr>
<tr>
<td>X</td>
<td>7C00000F4</td>
<td>B</td>
<td>popcntb</td>
<td></td>
<td>Population Count Bytes</td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>0400-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_addtb</td>
<td>Add Short Form</td>
</tr>
<tr>
<td>OIM5</td>
<td>2000-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_addt</td>
<td>Add Immediate Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>4600-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_andt[.]</td>
<td>AND Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>4500-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_andc</td>
<td>AND with Complement Long Word Form</td>
</tr>
<tr>
<td>IM5</td>
<td>2E00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_andi</td>
<td>AND Immediate Short Form</td>
</tr>
<tr>
<td>BD8</td>
<td>E800-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>be_b[.]</td>
<td>Branch [and Link]</td>
</tr>
<tr>
<td>BD8</td>
<td>E000-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bc</td>
<td>Branch Conditional Short Form</td>
</tr>
<tr>
<td>IM5</td>
<td>6000-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bclti</td>
<td>Bit Clear Immediate</td>
</tr>
<tr>
<td>C</td>
<td>0006-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bct[.]</td>
<td>Branch to Count Register [and Link]</td>
</tr>
<tr>
<td>IM5</td>
<td>6200-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bgeni</td>
<td>Bit Generate Immediate</td>
</tr>
<tr>
<td>C</td>
<td>0004-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_brl[.]</td>
<td>Branch to Link Register [and Link]</td>
</tr>
<tr>
<td>IM5</td>
<td>2C00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bmski</td>
<td>Bit Mask Generate Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6400-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bseti</td>
<td>Bit Set Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6600-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_bstsi</td>
<td>Bit Test Immediate</td>
</tr>
<tr>
<td>RR</td>
<td>0C00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_cmp</td>
<td>Compare Word</td>
</tr>
<tr>
<td>RR</td>
<td>0E00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_cmph</td>
<td>Compare Halfword Long Word Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>1F00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_cmph</td>
<td>Compare Halfword Logical Short Form</td>
</tr>
<tr>
<td>IM5</td>
<td>2A00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_cmpl</td>
<td>Compare Immediate Halfword Logical Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>0D00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_cmpl</td>
<td>Compare Logical Word</td>
</tr>
<tr>
<td>OIM5</td>
<td>2200-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_cmpli</td>
<td>Compare Logical Immediate Word</td>
</tr>
<tr>
<td>R</td>
<td>0000-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_extsb</td>
<td>Extend Sign Byte Short Form</td>
</tr>
<tr>
<td>R</td>
<td>00F0-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_extsh</td>
<td>Extend Sign Halfword Short Form</td>
</tr>
<tr>
<td>R</td>
<td>00C0-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_extzb</td>
<td>Extend Zero Byte</td>
</tr>
<tr>
<td>R</td>
<td>00E0-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_extzh</td>
<td>Extend Zero Halfword</td>
</tr>
<tr>
<td>C</td>
<td>0001-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_illegal</td>
<td>Illegal</td>
</tr>
<tr>
<td>C</td>
<td>0001-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_isync</td>
<td>Instruction Synchronize</td>
</tr>
<tr>
<td>SD4</td>
<td>8000-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_lzb</td>
<td>Load Byte and Zero Short Form</td>
</tr>
<tr>
<td>SD4</td>
<td>4A00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_lhz</td>
<td>Load Halfword and Zero Short Form</td>
</tr>
<tr>
<td>IM7</td>
<td>4800-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_li</td>
<td>Load Immediate Short Form</td>
</tr>
<tr>
<td>SD4</td>
<td>0C00-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_lwz</td>
<td>Load Word and Zero Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>0300-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mfar</td>
<td>Move from Alternate Register</td>
</tr>
<tr>
<td>R</td>
<td>00A0-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mfctr</td>
<td>Move From Count Register</td>
</tr>
<tr>
<td>R</td>
<td>0080-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mfr</td>
<td>Move From Link Register</td>
</tr>
<tr>
<td>RR</td>
<td>0100-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mr</td>
<td>Move Register</td>
</tr>
<tr>
<td>RR</td>
<td>0200-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mtar</td>
<td>Move To Alternate Register</td>
</tr>
<tr>
<td>R</td>
<td>00B0-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mtcgr</td>
<td>Move To Count Register</td>
</tr>
<tr>
<td>R</td>
<td>0090-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mfr</td>
<td>Move To Link Register</td>
</tr>
<tr>
<td>RR</td>
<td>0500-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_mullw</td>
<td>Multiply Low Word Short Form</td>
</tr>
<tr>
<td>R</td>
<td>0030-----</td>
<td>VLE</td>
<td></td>
<td></td>
<td>se_neg</td>
<td>Negate Short Form</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep.* Priv.</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>------------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>0020-----</td>
<td>VLE</td>
<td>se_not</td>
<td>NOT Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>4400-----</td>
<td>VLE</td>
<td>se_or</td>
<td>OR Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0009-----</td>
<td>H VLE</td>
<td>se_rfci</td>
<td>Return From Critical Interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>001A-----</td>
<td>H VLE</td>
<td>se_rfdi</td>
<td>Return From Debug Interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>000C-----</td>
<td>P VLE</td>
<td>se_rfgi</td>
<td>Return From Guest Interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>000B-----</td>
<td>P VLE</td>
<td>se_rfi</td>
<td>Return From Interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>4400-----</td>
<td>VLE</td>
<td>se_slw</td>
<td>Shift Left Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>4000-----</td>
<td>VLE</td>
<td>se_slwi</td>
<td>Shift Left Word Immediate Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM5</td>
<td>6C00-----</td>
<td>VLE</td>
<td>se_sraw</td>
<td>Shift Right Algebraic Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM5</td>
<td>6A00-----</td>
<td>VLE</td>
<td>se_srawi</td>
<td>Shift Right Algebraic Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>4000-----</td>
<td>VLE</td>
<td>se_srw</td>
<td>Shift Right Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM5</td>
<td>6800-----</td>
<td>VLE</td>
<td>se_srw</td>
<td>Shift Right Word Immediate Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>9000-----</td>
<td>VLE</td>
<td>se_stb</td>
<td>Store Byte Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>B000-----</td>
<td>VLE</td>
<td>se_sth</td>
<td>Store Halfword Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>D000-----</td>
<td>VLE</td>
<td>se_stw</td>
<td>Store Word Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>0600-----</td>
<td>VLE</td>
<td>se_sub</td>
<td>Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>0700-----</td>
<td>VLE</td>
<td>se_subf</td>
<td>Subtract From Short Form</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OIM5</td>
<td>2400-----</td>
<td>SR VLE</td>
<td>se_subi</td>
<td>Subtract Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000036 SR</td>
<td>64 sld[,]</td>
<td>Shift Left Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000030 SR</td>
<td>B sli[,]</td>
<td>Shift Left Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000634 SR</td>
<td>64 srad[,]</td>
<td>Shift Right Algebraic Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XS</td>
<td>7C000674 SR</td>
<td>64 srad[,]</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000630 SR</td>
<td>B sraw[,]</td>
<td>Shift Right Algebraic Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000670 SR</td>
<td>B sraw[,]</td>
<td>Shift Right Algebraic Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000436 SR</td>
<td>64 srd[,]</td>
<td>Shift Right Doubleword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000430 SR</td>
<td>B srw[,]</td>
<td>Shift Right Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00056D B</td>
<td>stbcx.</td>
<td>Store Byte Conditional Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000506 DS</td>
<td>stbdx</td>
<td>Store Byte with Decoration Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001BE P</td>
<td>E.PD stbepx</td>
<td>Store Byte by External Process ID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001EE B</td>
<td>stbux</td>
<td>Store Byte with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001AE B</td>
<td>stbx</td>
<td>Store Byte Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001AD DS</td>
<td>stdcx.</td>
<td>Store Doubleword Conditional Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005C6 DS</td>
<td>stddx</td>
<td>Store Doubleword with Decoration Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00013A P E.PD,64</td>
<td>stdepix</td>
<td>Store Doubleword by External Process ID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00016A</td>
<td>64 stdux</td>
<td>Store Doubleword with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00012A</td>
<td>64 stdx</td>
<td>Store Doubleword Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000746 DS</td>
<td>stfddx</td>
<td>Store Floating Doubleword with Decoration Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005BE P E.PD</td>
<td>stfdepix</td>
<td>Store Floating-Point Double by External Process ID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00072C B</td>
<td>sthbrx</td>
<td>Store Halfword Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005AD B</td>
<td>sthcx.</td>
<td>Store Halfword Conditional Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000546 DS</td>
<td>sthdx</td>
<td>Store Halfword with Decoration Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00033E P E.PD</td>
<td>sthepx</td>
<td>Store Halfword by External Process ID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00036E B</td>
<td>sthux</td>
<td>Store Halfword with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00032E B</td>
<td>sthx</td>
<td>Store Halfword Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005AA MA</td>
<td>stswi</td>
<td>Store String Word Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00052A MA</td>
<td>stswx</td>
<td>Store String Word Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00010E V</td>
<td>stvebx</td>
<td>Store Vector Element Byte Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00014E V</td>
<td>stvehx</td>
<td>Store Vector Element Halfword Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00064E P E.PD</td>
<td>stvexp</td>
<td>Store Vector by External Process ID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00060E P E.PD</td>
<td>stvexpx</td>
<td>Store Vector by External Process ID Indexed LRU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00018E V</td>
<td>stvewx</td>
<td>Store Vector Element Word Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001CE V</td>
<td>stvx</td>
<td>Store Vector Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003CE V</td>
<td>stvxl</td>
<td>Store Vector Indexed LRU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00052C B</td>
<td>stwbux</td>
<td>Store Word Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00012D B</td>
<td>stwcx.</td>
<td>Store Word Conditional Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000586 DS</td>
<td>stwdx</td>
<td>Store Word with Decoration Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00013E P E.PD</td>
<td>stwepx</td>
<td>Store Word by External Process ID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode</td>
<td>Dep.</td>
<td>Priv</td>
<td>Cat</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>7C00016E</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>stwux</td>
</tr>
<tr>
<td>X</td>
<td>7C00012E</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>stwx</td>
</tr>
<tr>
<td>XO</td>
<td>7C000050</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>subf[]</td>
</tr>
<tr>
<td>XO</td>
<td>7C000010</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>subf[]</td>
</tr>
<tr>
<td>XO</td>
<td>7C000110</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>subf[]</td>
</tr>
<tr>
<td>XO</td>
<td>7C0001D0</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>subfme[]</td>
</tr>
<tr>
<td>XO</td>
<td>7C000190</td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td>subfze[]</td>
</tr>
<tr>
<td>X</td>
<td>7C0004AC</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>sync</td>
</tr>
<tr>
<td>X</td>
<td>7C000088</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td>td</td>
</tr>
<tr>
<td>X</td>
<td>7C000624</td>
<td>H</td>
<td>E</td>
<td></td>
<td></td>
<td>tlbivax</td>
</tr>
<tr>
<td>X</td>
<td>7C000764</td>
<td>H</td>
<td>E</td>
<td></td>
<td></td>
<td>tlbre</td>
</tr>
<tr>
<td>X</td>
<td>7C000724</td>
<td>H</td>
<td>E</td>
<td></td>
<td></td>
<td>tlbsx</td>
</tr>
<tr>
<td>X</td>
<td>7C00046C</td>
<td>H</td>
<td>E</td>
<td></td>
<td></td>
<td>tlbwe</td>
</tr>
<tr>
<td>X</td>
<td>7C000008</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>tw</td>
</tr>
<tr>
<td>VX</td>
<td>10000180</td>
<td>V</td>
<td>vaddcuw</td>
<td></td>
<td></td>
<td>Vector Add and write Carry-out Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000000A</td>
<td>V</td>
<td>vaddfp</td>
<td></td>
<td></td>
<td>Vector Add Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>10000300</td>
<td>V</td>
<td>vaddsb</td>
<td></td>
<td></td>
<td>Vector Add Signed Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>10000340</td>
<td>V</td>
<td>vaddsds</td>
<td></td>
<td></td>
<td>Vector Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>10000380</td>
<td>V</td>
<td>vaddsws</td>
<td></td>
<td></td>
<td>Vector Add Signed Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>10000000</td>
<td>V</td>
<td>vaddubm</td>
<td></td>
<td></td>
<td>Vector Add Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>10000200</td>
<td>V</td>
<td>vaddubs</td>
<td></td>
<td></td>
<td>Vector Add Unsigned Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>10000040</td>
<td>V</td>
<td>vadduhm</td>
<td></td>
<td></td>
<td>Vector Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>10000240</td>
<td>V</td>
<td>vadduhw</td>
<td></td>
<td></td>
<td>Vector Add Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>10000080</td>
<td>V</td>
<td>vadduw</td>
<td></td>
<td></td>
<td>Vector Add Unsigned Word Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>10000280</td>
<td>V</td>
<td>vadduw</td>
<td></td>
<td></td>
<td>Vector Add Unsigned Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>10000404</td>
<td>V</td>
<td>vand</td>
<td></td>
<td></td>
<td>Vector Logical AND</td>
</tr>
<tr>
<td>VX</td>
<td>10000444</td>
<td>V</td>
<td>vandc</td>
<td></td>
<td></td>
<td>Vector Logical AND with Complement</td>
</tr>
<tr>
<td>VX</td>
<td>10000502</td>
<td>V</td>
<td>vavgsh</td>
<td></td>
<td></td>
<td>Vector Average Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>10000542</td>
<td>V</td>
<td>vavgsh</td>
<td></td>
<td></td>
<td>Vector Average Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000582</td>
<td>V</td>
<td>vavgsw</td>
<td></td>
<td></td>
<td>Vector Average Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000402</td>
<td>V</td>
<td>vavgub</td>
<td></td>
<td></td>
<td>Vector Average Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>10000442</td>
<td>V</td>
<td>vavguh</td>
<td></td>
<td></td>
<td>Vector Average Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000482</td>
<td>V</td>
<td>vavguw</td>
<td></td>
<td></td>
<td>Vector Average Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000034A</td>
<td>V</td>
<td>vctfx</td>
<td></td>
<td></td>
<td>Vector Convert From Signed Fixed-Point Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000300</td>
<td>V</td>
<td>vcfx</td>
<td></td>
<td></td>
<td>Vector Convert From</td>
</tr>
<tr>
<td>VC</td>
<td>100003C6</td>
<td>V</td>
<td>vcmpsfp[.]</td>
<td></td>
<td></td>
<td>Vector Compare Bounds Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>100000C6</td>
<td>V</td>
<td>vcmpqfp[.]</td>
<td></td>
<td></td>
<td>Vector Compare Equal To Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>10000006</td>
<td>V</td>
<td>vcmpqub[.]</td>
<td></td>
<td></td>
<td>Vector Compare Equal To Unsigned Byte</td>
</tr>
<tr>
<td>VC</td>
<td>10000046</td>
<td>V</td>
<td>vcmpquh[.]</td>
<td></td>
<td></td>
<td>Vector Compare Equal To Unsigned Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>10000086</td>
<td>V</td>
<td>vcmpquw[.]</td>
<td></td>
<td></td>
<td>Vector Compare Equal To Unsigned Word</td>
</tr>
<tr>
<td>VC</td>
<td>100001C6</td>
<td>V</td>
<td>vcmpqepf[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than or Equal To Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>100002C6</td>
<td>V</td>
<td>vcmpgtfp[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>10000306</td>
<td>V</td>
<td>vcmpgtbsb[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Signed Byte</td>
</tr>
<tr>
<td>VC</td>
<td>10000346</td>
<td>V</td>
<td>vcmpgtssh[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Signed Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>10000386</td>
<td>V</td>
<td>vcmpgtsw[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Signed Word</td>
</tr>
<tr>
<td>VC</td>
<td>10000206</td>
<td>V</td>
<td>vcmpgtub[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Unsigned Byte</td>
</tr>
<tr>
<td>VC</td>
<td>10000246</td>
<td>V</td>
<td>vcmpgtuh[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Unsigned Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>10000286</td>
<td>V</td>
<td>vcmpgtuw[.]</td>
<td></td>
<td></td>
<td>Vector Compare Greater Than Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>100003CA</td>
<td>V</td>
<td>vctfxs</td>
<td></td>
<td></td>
<td>Vector Convert To Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>1000038A</td>
<td>V</td>
<td>vcuxt</td>
<td></td>
<td></td>
<td>Vector Convert To Unsigned Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>1000018A</td>
<td>V</td>
<td>vexpte</td>
<td></td>
<td></td>
<td>Vector 2 Raised to the Exponent Estimate Floating-Point</td>
</tr>
<tr>
<td>VX</td>
<td>100001CA</td>
<td>V</td>
<td>vlogefp</td>
<td></td>
<td></td>
<td>Vector Log Base 2 Estimate Floating-Point</td>
</tr>
<tr>
<td>VA</td>
<td>100002E</td>
<td>V</td>
<td>vmaddfp</td>
<td></td>
<td></td>
<td>Vector Multiply-Add Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>100004OA</td>
<td>V</td>
<td>vmmaxfp</td>
<td></td>
<td></td>
<td>Vector Maximum Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>10000102</td>
<td>V</td>
<td>vmmaxb</td>
<td></td>
<td></td>
<td>Vector Maximum Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>10000142</td>
<td>V</td>
<td>vmmaxh</td>
<td></td>
<td></td>
<td>Vector Maximum Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000182</td>
<td>V</td>
<td>vmmaxw</td>
<td></td>
<td></td>
<td>Vector Maximum Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000002</td>
<td>V</td>
<td>vmxub</td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>10000042</td>
<td>V</td>
<td>vmxuh</td>
<td></td>
<td></td>
<td>Vector Maximum Unsigned Halfword</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)2</td>
<td>Mode Dep.1</td>
<td>Cat1</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000082</td>
<td>V</td>
<td></td>
<td>vmaxuw</td>
<td>Vector Maximum Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000020</td>
<td>V</td>
<td></td>
<td>vmhaddshs</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000021</td>
<td>V</td>
<td></td>
<td>vmraddshs</td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000044A</td>
<td>V</td>
<td></td>
<td>vminfp</td>
<td>Vector Minimum Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002C</td>
<td>V</td>
<td></td>
<td>vminsb</td>
<td>Vector Minimum Signed Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002D</td>
<td>V</td>
<td></td>
<td>vminsh</td>
<td>Vector Minimum Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000382</td>
<td>V</td>
<td></td>
<td>vminsw</td>
<td>Vector Minimum Signed Word</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002F</td>
<td>V</td>
<td></td>
<td>vminub</td>
<td>Vector Minimum Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002G</td>
<td>V</td>
<td></td>
<td>vminuh</td>
<td>Vector Minimum Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000282</td>
<td>V</td>
<td></td>
<td>vminuw</td>
<td>Vector Minimum Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000022</td>
<td>V</td>
<td></td>
<td>vmladuhm</td>
<td>Vector Multiply-Low-Add Signed Halfword Modulo</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000000C</td>
<td>V</td>
<td></td>
<td>vmrghb</td>
<td>Vector Merge High Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000004C</td>
<td>V</td>
<td></td>
<td>vmrghh</td>
<td>Vector Merge High Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000008C</td>
<td>V</td>
<td></td>
<td>vmrghw</td>
<td>Vector Merge High Word</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000010C</td>
<td>V</td>
<td></td>
<td>vmgib</td>
<td>Vector Merge Low Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000014C</td>
<td>V</td>
<td></td>
<td>vmgih</td>
<td>Vector Merge Low Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000018C</td>
<td>V</td>
<td></td>
<td>vmglw</td>
<td>Vector Merge Low Word</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000025</td>
<td>V</td>
<td></td>
<td>vmsummbm</td>
<td>Vector Multiply-Sum Mixed Byte Modulo</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000026</td>
<td>V</td>
<td></td>
<td>vmsumshs</td>
<td>Vector Multiply-Sum Signed Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000027</td>
<td>V</td>
<td></td>
<td>vmsumubm</td>
<td>Vector Multiply-Sum Unsigned Byte Modulo</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000028</td>
<td>V</td>
<td></td>
<td>vmsumuhm</td>
<td>Vector Multiply-Sum Unsigned Halfword Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000002A</td>
<td>V</td>
<td></td>
<td>vmulesb</td>
<td>Vector Multiply Even Signed Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002B</td>
<td>V</td>
<td></td>
<td>vmulesh</td>
<td>Vector Multiply Even Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002C</td>
<td>V</td>
<td></td>
<td>vmuleub</td>
<td>Vector Multiply Even Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002D</td>
<td>V</td>
<td></td>
<td>vmuleuh</td>
<td>Vector Multiply Even Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000001B</td>
<td>V</td>
<td></td>
<td>vmlsh</td>
<td>Vector Multiply Odd Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000001C</td>
<td>V</td>
<td></td>
<td>vmlsho</td>
<td>Vector Multiply Odd Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000001D</td>
<td>V</td>
<td></td>
<td>vmlsho</td>
<td>Vector Multiply Odd Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000001E</td>
<td>V</td>
<td></td>
<td>vmlso</td>
<td>Vector Multiply Odd Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002F</td>
<td>V</td>
<td></td>
<td>vmsubfp</td>
<td>Vector Negative Multiply-Subtract Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000032</td>
<td>V</td>
<td></td>
<td>vno</td>
<td>Vector Logical NOR</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000034</td>
<td>V</td>
<td></td>
<td>vnor</td>
<td>Vector Logical OR</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000038</td>
<td>V</td>
<td></td>
<td>vperm</td>
<td>Vector Permute</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000030E</td>
<td>V</td>
<td></td>
<td>vpkpx</td>
<td>Vector Pack Pixel</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000018E</td>
<td>V</td>
<td></td>
<td>vpkshss</td>
<td>Vector Pack Signed Halfword Signed Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000019E</td>
<td>V</td>
<td></td>
<td>vpkshus</td>
<td>Vector Pack Signed Halfword Unsaturated</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000001CE</td>
<td>V</td>
<td></td>
<td>vpkswss</td>
<td>Vector Pack Signed Word Signed Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000001DE</td>
<td>V</td>
<td></td>
<td>vpkswus</td>
<td>Vector Pack Signed Word Unsaturated</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000000E</td>
<td>V</td>
<td></td>
<td>vpkuhms</td>
<td>Vector Pack Signed Halfword Unsaturated</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000000F</td>
<td>V</td>
<td></td>
<td>vpkwhms</td>
<td>Vector Pack Signed Halfword Unsaturated</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000001A</td>
<td>V</td>
<td></td>
<td>vpkwus</td>
<td>Vector Pack Signed Word Unsaturated</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000010A</td>
<td>V</td>
<td></td>
<td>vrefep</td>
<td>Vector Reciprocal Estimate Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000002CA</td>
<td>V</td>
<td></td>
<td>vrfim</td>
<td>Vector Round to Single-Precision Integer toward -Infinity</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000002DA</td>
<td>V</td>
<td></td>
<td>vrfin</td>
<td>Vector Round to Single-Precision Integer Nearest</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000028A</td>
<td>V</td>
<td></td>
<td>vrfip</td>
<td>Vector Round to Single-Precision Integer toward +Infinity</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000024A</td>
<td>V</td>
<td></td>
<td>vrfiz</td>
<td>Vector Round to Single-Precision Integer toward Zero</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000004</td>
<td>V</td>
<td></td>
<td>vrlb</td>
<td>Vector Rotate Left Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000044</td>
<td>V</td>
<td></td>
<td>vrlh</td>
<td>Vector Rotate Left Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000084</td>
<td>V</td>
<td></td>
<td>vrlw</td>
<td>Vector Rotate Left Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000014A</td>
<td>V</td>
<td></td>
<td>vrsqrtfep</td>
<td>Vector Reciprocal Square Root Estimate Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002A</td>
<td>V</td>
<td></td>
<td>vsel</td>
<td>Vector Select</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>1000001C4</td>
<td>V</td>
<td></td>
<td>vsl</td>
<td>Vector Shift Left</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000104</td>
<td>V</td>
<td></td>
<td>vslb</td>
<td>Vector Shift Left Byte</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000002C</td>
<td>V</td>
<td></td>
<td>vslidoi</td>
<td>Vector Shift Left Double by Octet Immediate</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>100000144</td>
<td>V</td>
<td></td>
<td>vsl</td>
<td>Vector Shift Left Halfword</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>10000040C</td>
<td>V</td>
<td></td>
<td>vsl</td>
<td>Vector Shift Left by Octet</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100000184</td>
<td>V</td>
<td></td>
<td>vslw</td>
<td>Vector Shift Left Word</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix A. VLE Instruction Set Sorted by Mnemonic

**Form** | **Opcode (hexadecimal)\(^2\)** | **Mode Dep.\(^1\)** | **Priv\(^1\)** | **Cat\(^1\)** | **Mnemonic** | **Instruction**
--- | --- | --- | --- | --- | --- | ---
VX | 1000020C | V |  |  | vsplitb | Vector Splat Byte
VX | 1000024C | V |  |  | vsplitlth | Vector Splat Halfword
VX | 1000030C | V |  |  | vsplitlsb | Vector Splat Immediate Signed Byte
VX | 1000034C | V |  |  | vsplilsh | Vector Splat Immediate Signed Halfword
VX | 1000038C | V |  |  | vsplisw | Vector Splat Immediate Signed Word
VX | 1000028C | V |  |  | vsplthw | Vector Splat Word
VX | 100002C4 | V |  |  | vsr | Vector Shift Right
VX | 10000304 | V |  |  | vsrab | Vector Shift Right Algebraic Word
VX | 10000344 | V |  |  | vsrah | Vector Shift Right Algebraic Halfword
VX | 10000384 | V |  |  | vsraw | Vector Shift Right Algebraic Word
VX | 10000204 | V |  |  | vsrb | Vector Shift Right Byte
VX | 10000244 | V |  |  | vsrh | Vector Shift Right Halfword
VX | 1000044C | V |  |  | vsro | Vector Shift Right by Octet
VX | 10000284 | V |  |  | vsrw | Vector Shift Right Word
VX | 10000580 | V |  |  | vsubcuw | Vector Subtract and Write Carry-Out Unsigned Word
VX | 100004A | V |  |  | vsubfp | Vector Subtract Single-Precision
VX | 10000700 | V |  |  | vsubsbs | Vector Subtract Signed Byte Saturate
VX | 10000740 | V |  |  | vsubshs | Vector Subtract Signed Halfword Saturate
VX | 10000400 | V |  |  | vsubsws | Vector Subtract Signed Word Saturate
VX | 10000600 | V |  |  | vsububs | Vector Subtract Unsigned Byte Saturate
VX | 10000440 | V |  |  | vsubuhm | Vector Subtract Unsigned Halfword Modulo
VX | 10000640 | V |  |  | vsuhbs | Vector Subtract Unsigned Halfword Saturate
VX | 10000480 | V |  |  | vsuwm | Vector Subtract Unsigned Word Modulo
VX | 10000680 | V |  |  | vsubws | Vector SubtractUnsigned Word Saturate
VX | 10000688 | V |  |  | vsum2sws | Vector Sum across Half Signed Word Saturate
VX | 10000708 | V |  |  | vsum4sbs | Vector Sum across Quarter Signed Byte Saturate
VX | 10000648 | V |  |  | vsum4shs | Vector Sum across Quarter Signed Halfword Saturate
VX | 10000608 | V |  |  | vsum4ubs | Vector Sum across Quarter Unsigned Byte Saturate
VX | 10000788 | V |  |  | vsumsws | Vector Sum across Signed Word Saturate
VX | 1000034E | V |  |  | vupkpx | Vector Unpack High Pixel
VX | 1000020E | V |  |  | vupkshs | Vector Unpack High Signed Byte
VX | 1000024E | V |  |  | vupksh | Vector Unpack High Signed Halfword
VX | 100003CE | V |  |  | vupklpx | Vector Unpack Low Pixel
VX | 1000028E | V |  |  | vupklsb | Vector Unpack Low Signed Byte
VX | 100002CE | V |  |  | vupks | Vector Unpack Low Signed Halfword
VX | 100004C4 | V |  |  | vxor | Vector Logical XOR
X | 7C000007C | P |  |  | wt | wait
X | 7C0000106 | P |  |  | wrtee | Write MSR External Enable
X | 7C0000146 | P |  |  | wrteei | Write MSR External Enable Immediate
X | 7C0000278 | SR | B |  | xor[.] | XOR

---

1. See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.

2. For 16-bit instructions, the “Opcode” column represents the 16-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits; dashes are used following the opcode to indicate the form is a 16-bit instruction. For 32-bit instructions, the “Opcode” column represents the 32-bit hexadecimal instruction encoding with the opcode, extended opcode, and other fields with fixed values in the corresponding fields in the instruction, and with 0...s in bit positions which are not opcode, extended opcode or fixed value bits.
Appendix B. VLE Instruction Set Sorted by Opcode

This appendix lists all the instructions available in VLE mode in the Power ISA, in order by opcode. Opcodes that are not defined below are treated as illegal by category VLE.

<table>
<thead>
<tr>
<th>Form</th>
<th>Opcode (hexadecimal)</th>
<th>Mode Dep Priv</th>
<th>Cat1</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0000-----</td>
<td></td>
<td>VLE</td>
<td>se_illegal</td>
<td>Illegal</td>
</tr>
<tr>
<td>C</td>
<td>0001-----</td>
<td></td>
<td>VLE</td>
<td>se_isync</td>
<td>Instruction Synchronize</td>
</tr>
<tr>
<td>C</td>
<td>0002-----</td>
<td></td>
<td>VLE</td>
<td>se_sc</td>
<td>System Call</td>
</tr>
<tr>
<td>C</td>
<td>0004-----</td>
<td>P</td>
<td>VLE</td>
<td>se_brl[]</td>
<td>Branch to Link Register [and Link]</td>
</tr>
<tr>
<td>C</td>
<td>0006-----</td>
<td>H</td>
<td>VLE</td>
<td>se_bctr[]</td>
<td>Branch to Count Register [and Link]</td>
</tr>
<tr>
<td>C</td>
<td>0008-----</td>
<td>H</td>
<td>VLE</td>
<td>se_fci</td>
<td>Return From Interrupt</td>
</tr>
<tr>
<td>C</td>
<td>0009-----</td>
<td>H</td>
<td>VLE</td>
<td>se_fdi</td>
<td>Return From Debug Interrupt</td>
</tr>
<tr>
<td>C</td>
<td>000A-----</td>
<td>H</td>
<td>VLE</td>
<td>se_rmcni</td>
<td>Return From Machine Check Interrupt</td>
</tr>
<tr>
<td>C</td>
<td>000B-----</td>
<td>H</td>
<td>VLE</td>
<td>se_rfi</td>
<td>Return From Guest Interrupt</td>
</tr>
<tr>
<td>C</td>
<td>000C-----</td>
<td>P</td>
<td>VLE</td>
<td>se_rfi</td>
<td>Return From Interrupt</td>
</tr>
<tr>
<td>R</td>
<td>0020-----</td>
<td></td>
<td>VLE</td>
<td>se_not</td>
<td>NOT Short Form</td>
</tr>
<tr>
<td>R</td>
<td>0030-----</td>
<td></td>
<td>VLE</td>
<td>se_neg</td>
<td>Negate Short Form</td>
</tr>
<tr>
<td>R</td>
<td>0080-----</td>
<td></td>
<td>VLE</td>
<td>se_mfr</td>
<td>Move From Link Register</td>
</tr>
<tr>
<td>R</td>
<td>0090-----</td>
<td></td>
<td>VLE</td>
<td>se_mtr</td>
<td>Move To Link Register</td>
</tr>
<tr>
<td>R</td>
<td>00A0-----</td>
<td></td>
<td>VLE</td>
<td>se_mcfr</td>
<td>Move From Count Register</td>
</tr>
<tr>
<td>R</td>
<td>00B0-----</td>
<td></td>
<td>VLE</td>
<td>se_mCfr</td>
<td>Move To Count Register</td>
</tr>
<tr>
<td>R</td>
<td>00C0-----</td>
<td></td>
<td>VLE</td>
<td>se_extzb</td>
<td>Extend Zero Byte</td>
</tr>
<tr>
<td>R</td>
<td>00D0-----</td>
<td></td>
<td>VLE</td>
<td>se_extsb</td>
<td>Extend Sign Byte Short Form</td>
</tr>
<tr>
<td>R</td>
<td>00E0-----</td>
<td></td>
<td>VLE</td>
<td>se_extzh</td>
<td>Extend Zero Halfword</td>
</tr>
<tr>
<td>R</td>
<td>0100-----</td>
<td></td>
<td>VLE</td>
<td>se_mfr</td>
<td>Move From Link Register</td>
</tr>
<tr>
<td>R</td>
<td>01F0-----</td>
<td></td>
<td>VLE</td>
<td>se_mtr</td>
<td>Move To Link Register</td>
</tr>
<tr>
<td>RR</td>
<td>0200-----</td>
<td></td>
<td>VLE</td>
<td>se_mfr</td>
<td>Move From Alternate Register</td>
</tr>
<tr>
<td>RR</td>
<td>0300-----</td>
<td></td>
<td>VLE</td>
<td>se_mfar</td>
<td>Move from Alternate Register</td>
</tr>
<tr>
<td>RR</td>
<td>0400-----</td>
<td></td>
<td>VLE</td>
<td>se_add</td>
<td>Add Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>0500-----</td>
<td></td>
<td>VLE</td>
<td>se_mulw</td>
<td>Multiply Low Word Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>0600-----</td>
<td></td>
<td>VLE</td>
<td>se_sub</td>
<td>Subtract</td>
</tr>
<tr>
<td>RR</td>
<td>0700-----</td>
<td></td>
<td>VLE</td>
<td>se_subf</td>
<td>Subtract From Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>0C00-----</td>
<td></td>
<td>VLE</td>
<td>se_cmp</td>
<td>Compare Word</td>
</tr>
<tr>
<td>RR</td>
<td>0D00-----</td>
<td></td>
<td>VLE</td>
<td>se_cmpl</td>
<td>Compare Logical Word</td>
</tr>
<tr>
<td>RR</td>
<td>0E00-----</td>
<td></td>
<td>VLE</td>
<td>se_cmph</td>
<td>Compare Halfword Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>0F00-----</td>
<td></td>
<td>VLE</td>
<td>se_cmphl</td>
<td>Compare Halfword Logical Short Form</td>
</tr>
<tr>
<td>VX</td>
<td>10000001</td>
<td>V</td>
<td>vaddubm</td>
<td>Vector Add Unsigned Byte Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000002</td>
<td>V</td>
<td>vmaxub</td>
<td>Vector Maximum Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000004</td>
<td>V</td>
<td>vrlb</td>
<td>Vector Rotate Left Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000006</td>
<td>V</td>
<td>vcmpqub[.]</td>
<td>Vector Compare Equal To Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000008</td>
<td>V</td>
<td>vmuloub</td>
<td>Vector Multiply Odd Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000000A</td>
<td>V</td>
<td>vaddfp</td>
<td>Vector Add Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000000C</td>
<td>V</td>
<td>vmrphb</td>
<td>Vector Merge High Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000000E</td>
<td>V</td>
<td>vpkuhum</td>
<td>Vector Pack Unsigned Halfword Unsigned Modulo</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>10000010</td>
<td>SR</td>
<td>LMA</td>
<td>mulhwu[.]</td>
<td>Multiply High Halfword to Word Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>10000018</td>
<td>SR</td>
<td>LMA</td>
<td>machhwu[0][.]</td>
<td>Multiply Accumulate High Halfword to Word Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>10000020</td>
<td>V</td>
<td>vmhaddshs</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)(^2)</td>
<td>Mode Dep Priv</td>
<td>Cat(^1)</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------</td>
<td>---------------</td>
<td>-----------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VA</td>
<td>100000021</td>
<td>V</td>
<td></td>
<td>vmnaddshs</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA</td>
<td>100000022</td>
<td>V</td>
<td></td>
<td>vminadduhm</td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>100000024</td>
<td>V</td>
<td></td>
<td>vmsumubm</td>
<td>Vector Multiply-Sum Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>100000025</td>
<td>V</td>
<td></td>
<td>vmsummbm</td>
<td>Vector Multiply-Sum Mixed Byte Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>100000026</td>
<td>V</td>
<td></td>
<td>vmsumuhm</td>
<td>Vector Multiply-Sum Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>100000027</td>
<td>V</td>
<td></td>
<td>vmsumuhs</td>
<td>Vector Multiply-Sum Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>VA</td>
<td>100000028</td>
<td>V</td>
<td></td>
<td>vmsumshm</td>
<td>Vector Multiply-Sum Signed Halfword Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>100000029</td>
<td>V</td>
<td></td>
<td>vmsumshs</td>
<td>Vector Multiply-Sum Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA</td>
<td>10000002A</td>
<td>V</td>
<td></td>
<td>vsel</td>
<td>Vector Select</td>
</tr>
<tr>
<td>VA</td>
<td>10000002B</td>
<td>V</td>
<td></td>
<td>vperm</td>
<td>Vector Permute</td>
</tr>
<tr>
<td>VA</td>
<td>10000002C</td>
<td>V</td>
<td></td>
<td>vssldoi</td>
<td>Vector Shift Left Double by Octet Immediate</td>
</tr>
<tr>
<td>VA</td>
<td>10000002E</td>
<td>V</td>
<td></td>
<td>vmmaddfp</td>
<td>Vector Multiply-Add Single-Precision</td>
</tr>
<tr>
<td>VA</td>
<td>10000002F</td>
<td>V</td>
<td></td>
<td>vmmnsubfp</td>
<td>Vector Negative Multiply-Subtract Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>100000040</td>
<td>V</td>
<td></td>
<td>vadduhm</td>
<td>Vector Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>100000042</td>
<td>V</td>
<td></td>
<td>vmauxh</td>
<td>Vector Maximum Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>100000044</td>
<td>V</td>
<td></td>
<td>vrlh</td>
<td>Vector Rotate Left Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>100000046</td>
<td>V</td>
<td></td>
<td>vccmpequh[]</td>
<td>Vector Compare Equal To Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>100000048</td>
<td>V</td>
<td></td>
<td>vmuioh</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>100000048A</td>
<td>V</td>
<td></td>
<td>vsubfp</td>
<td>Vector Subtract Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>10000004C</td>
<td>V</td>
<td></td>
<td>vmrgfh</td>
<td>Vector Merge High Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000004E</td>
<td>V</td>
<td></td>
<td>vwpkuwum</td>
<td>Vector Pack Unsigned Word Modulo</td>
</tr>
<tr>
<td>X</td>
<td>100000050</td>
<td>SR</td>
<td>LMA</td>
<td>mulhhw[,]</td>
<td>Multiply High Halfword to Word Signed</td>
</tr>
<tr>
<td>XO</td>
<td>100000058</td>
<td>SR</td>
<td>LMA</td>
<td>machhw[o][.]</td>
<td>Multiply Accumulate High Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>10000005C</td>
<td>SR</td>
<td>LMA</td>
<td>nmachhw[o][.]</td>
<td>Negative Multiply Accumulate High Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>VX</td>
<td>100000080</td>
<td>V</td>
<td></td>
<td>vadduwm</td>
<td>Vector Add Unsigned Word Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>100000082</td>
<td>V</td>
<td></td>
<td>vmauxw</td>
<td>Vector Maximum Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>100000084</td>
<td>V</td>
<td></td>
<td>vrlw</td>
<td>Vector Rotate Left Word</td>
</tr>
<tr>
<td>VC</td>
<td>100000086</td>
<td>V</td>
<td></td>
<td>vccmpequw[,]</td>
<td>Vector Compare Equal To Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000008C</td>
<td>V</td>
<td></td>
<td>vmrgfw</td>
<td>Vector Merge High Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000008E</td>
<td>V</td>
<td></td>
<td>vwpkuhus</td>
<td>Vector Pack Unrounded Halfword Unrounded Saturate</td>
</tr>
<tr>
<td>XO</td>
<td>100000098</td>
<td>SR</td>
<td>LMA</td>
<td>machwsu[o][.]</td>
<td>Multiply Accumulate High Halfword to Word Saturate</td>
</tr>
<tr>
<td>VC</td>
<td>1000000C6</td>
<td>V</td>
<td></td>
<td>vccmpeqp[,]</td>
<td>Vector Compare Equal To Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>1000000CE</td>
<td>V</td>
<td></td>
<td>vwpkuwus</td>
<td>Vector Pack Unrounded Word Unrounded Saturate</td>
</tr>
<tr>
<td>XO</td>
<td>1000000D8</td>
<td>SR</td>
<td>LMA</td>
<td>machws[o][.]</td>
<td>Multiply Accumulate High Halfword to Word Saturate</td>
</tr>
<tr>
<td>XO</td>
<td>1000000DC</td>
<td>SR</td>
<td>LMA</td>
<td>nmachws[o][.]</td>
<td>Negative Multiply Accumulate High Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>VX</td>
<td>10000102</td>
<td>V</td>
<td></td>
<td>vmaxsb</td>
<td>Vector Maximum Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>10000104</td>
<td>V</td>
<td></td>
<td>vsslb</td>
<td>Vector Shift Left Byte</td>
</tr>
<tr>
<td>VX</td>
<td>10000108</td>
<td>V</td>
<td></td>
<td>vmulosb</td>
<td>Vector Multiply Odd Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000010A</td>
<td>V</td>
<td></td>
<td>vreffp</td>
<td>Vector Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>1000010C</td>
<td>V</td>
<td></td>
<td>vmrgeb</td>
<td>Vector Merge Low Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000010E</td>
<td>V</td>
<td></td>
<td>vwpkshus</td>
<td>Vector Pack Signed Halfword Unrounded Saturate</td>
</tr>
<tr>
<td>X</td>
<td>10000110</td>
<td>SR</td>
<td>LMA</td>
<td>mulchwu[,]</td>
<td>Multiply Cross Halfword to Word Unrounded</td>
</tr>
<tr>
<td>XO</td>
<td>10000118</td>
<td>SR</td>
<td>LMA</td>
<td>machwu[o][.]</td>
<td>Multiply Accumulate Cross Halfword to Word Modulo Unrounded</td>
</tr>
<tr>
<td>VX</td>
<td>10000142</td>
<td>V</td>
<td></td>
<td>vmaxsgh</td>
<td>Vector Maximum Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000144</td>
<td>V</td>
<td></td>
<td>vsslh</td>
<td>Vector Shift Left Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000148</td>
<td>V</td>
<td></td>
<td>vmulosgh</td>
<td>Vector Multiply Odd Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>1000014A</td>
<td>V</td>
<td></td>
<td>vrsqrtfp</td>
<td>Vector Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>1000014C</td>
<td>V</td>
<td></td>
<td>vmrglh</td>
<td>Vector Merge Low Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>1000015E</td>
<td>V</td>
<td></td>
<td>vwpkshwus</td>
<td>Vector Pack Signed Word Unrounded Saturate</td>
</tr>
<tr>
<td>X</td>
<td>10000150</td>
<td>SR</td>
<td>LMA</td>
<td>mulchwu[,]</td>
<td>Multiply Cross Halfword to Word Unrounded</td>
</tr>
<tr>
<td>XO</td>
<td>10000158</td>
<td>SR</td>
<td>LMA</td>
<td>machwu[o][.]</td>
<td>Multiply Accumulate Cross Halfword to Word Modulo Unrounded</td>
</tr>
<tr>
<td>XO</td>
<td>1000015C</td>
<td>SR</td>
<td>LMA</td>
<td>nmachwu[o][.]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Modulo Signed</td>
</tr>
</tbody>
</table>

1332  Power ISA™ - Book VLE
<table>
<thead>
<tr>
<th>Form</th>
<th>Opcode (hexadecimal)</th>
<th>Mode Dep. Priv.</th>
<th>Cat¹</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>VX</td>
<td>00001B0</td>
<td>V</td>
<td></td>
<td>vaddcuw</td>
<td>Vector Add and write Carry-out Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>00001B2</td>
<td>V</td>
<td></td>
<td>vmaxsw</td>
<td>Vector Maximum Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>00001B4</td>
<td>V</td>
<td></td>
<td>vslw</td>
<td>Vector Shift Left Word</td>
</tr>
<tr>
<td>VX</td>
<td>00001B6</td>
<td>V</td>
<td></td>
<td>vexptefp</td>
<td>Vector 2 Raised to the Exponent Estimate Floating-Point</td>
</tr>
<tr>
<td>VX</td>
<td>00001B8</td>
<td>V</td>
<td></td>
<td>vmrlo0</td>
<td>Vector Merge Low Word</td>
</tr>
<tr>
<td>VX</td>
<td>00001B8E</td>
<td>V</td>
<td></td>
<td>vpkshe0</td>
<td>Vector Pack Signed Halfword Signed Saturate</td>
</tr>
<tr>
<td>XO</td>
<td>0000198</td>
<td>SR</td>
<td></td>
<td>macchwsu0</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Unsigned</td>
</tr>
<tr>
<td>VX</td>
<td>00001C4</td>
<td>V</td>
<td></td>
<td>vsl</td>
<td>Vector Shift Left</td>
</tr>
<tr>
<td>VC</td>
<td>00001C6</td>
<td>V</td>
<td></td>
<td>vcmpgefp</td>
<td>Vector Compare Greater Than or Equal To Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>00001CA</td>
<td>V</td>
<td></td>
<td>vlogefp</td>
<td>Vector Log Base 2 Estimate Floating-Point</td>
</tr>
<tr>
<td>VX</td>
<td>00001CE</td>
<td>V</td>
<td></td>
<td>vpksws</td>
<td>Vector Pack Signed Word Signed Saturate</td>
</tr>
<tr>
<td>XO</td>
<td>00001D8</td>
<td>SR</td>
<td></td>
<td>macchws0</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>00001DC</td>
<td>SR</td>
<td></td>
<td>nmacchws0</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>EVX</td>
<td>0000200</td>
<td>SP</td>
<td></td>
<td>evaddw</td>
<td>Vector Add Word</td>
</tr>
<tr>
<td>VX</td>
<td>0000200</td>
<td>V</td>
<td></td>
<td>vaddubs</td>
<td>Vector Add Unsigned Byte Saturate</td>
</tr>
<tr>
<td>EVX</td>
<td>0000202</td>
<td>SP</td>
<td></td>
<td>evaddw</td>
<td>Vector Add Immediate Word</td>
</tr>
<tr>
<td>VX</td>
<td>0000202</td>
<td>V</td>
<td></td>
<td>vminub</td>
<td>Vector Minimum Unsigned Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>0000204</td>
<td>SP</td>
<td></td>
<td>evsubfw</td>
<td>Vector Subtract from Word</td>
</tr>
<tr>
<td>VX</td>
<td>0000204</td>
<td>V</td>
<td></td>
<td>vsrb</td>
<td>Vector Shift Right Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>0000206</td>
<td>SP</td>
<td></td>
<td>evsubifw</td>
<td>Vector Subtract Immediate from Word</td>
</tr>
<tr>
<td>VC</td>
<td>0000206</td>
<td>V</td>
<td></td>
<td>vcmpgtub</td>
<td>Vector Compare Greater Than Unsigned Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>0000208</td>
<td>SP</td>
<td></td>
<td>evabs</td>
<td>Vector Absolute Value</td>
</tr>
<tr>
<td>VX</td>
<td>0000208</td>
<td>V</td>
<td></td>
<td>vmuleub</td>
<td>Vector Multiply Even Unsigned Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>0000209</td>
<td>SP</td>
<td></td>
<td>evneg</td>
<td>Vector Negate</td>
</tr>
<tr>
<td>EVX</td>
<td>000020A</td>
<td>SP</td>
<td></td>
<td>evextsb</td>
<td>Vector Extend Sign Byte</td>
</tr>
<tr>
<td>VX</td>
<td>000020A</td>
<td>V</td>
<td></td>
<td>vrfin</td>
<td>Vector Round to Single-Precision Integer Nearest</td>
</tr>
<tr>
<td>EVX</td>
<td>000020B</td>
<td>SP</td>
<td></td>
<td>evextsh</td>
<td>Vector Extend Sign Halfword</td>
</tr>
<tr>
<td>EVX</td>
<td>000020C</td>
<td>SP</td>
<td></td>
<td>evrndw</td>
<td>Vector Round Word</td>
</tr>
<tr>
<td>VX</td>
<td>000020C</td>
<td>V</td>
<td></td>
<td>vspltb</td>
<td>Vector Splat Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>000020D</td>
<td>SP</td>
<td></td>
<td>evcntlzwb</td>
<td>Vector Count Leading Zeros Word</td>
</tr>
<tr>
<td>EVX</td>
<td>000020E</td>
<td>SP</td>
<td></td>
<td>evcntlsw</td>
<td>Vector Count Leading Signed Bits Word</td>
</tr>
<tr>
<td>VX</td>
<td>000020E</td>
<td>V</td>
<td></td>
<td>vupksbsb</td>
<td>Vector Unpack High Signed Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>000020F</td>
<td>SP</td>
<td></td>
<td>brinc</td>
<td>Bit Reversed Increment</td>
</tr>
<tr>
<td>EVX</td>
<td>0000211</td>
<td>SP</td>
<td></td>
<td>evand</td>
<td>Vector AND</td>
</tr>
<tr>
<td>EVX</td>
<td>0000212</td>
<td>SP</td>
<td></td>
<td>evandc</td>
<td>Vector AND with Complement</td>
</tr>
<tr>
<td>EVX</td>
<td>0000216</td>
<td>SP</td>
<td></td>
<td>evxor</td>
<td>Vector XOR</td>
</tr>
<tr>
<td>EVX</td>
<td>0000217</td>
<td>SP</td>
<td></td>
<td>evor</td>
<td>Vector OR</td>
</tr>
<tr>
<td>EVX</td>
<td>0000218</td>
<td>SP</td>
<td></td>
<td>evnor</td>
<td>Vector NOR</td>
</tr>
<tr>
<td>EVX</td>
<td>0000219</td>
<td>SP</td>
<td></td>
<td>eveqv</td>
<td>Vector Equivalent</td>
</tr>
<tr>
<td>EVX</td>
<td>000021B</td>
<td>SP</td>
<td></td>
<td>evorc</td>
<td>Vector OR with Complement</td>
</tr>
<tr>
<td>EVX</td>
<td>000021E</td>
<td>SP</td>
<td></td>
<td>evnand</td>
<td>Vector NAND</td>
</tr>
<tr>
<td>EVX</td>
<td>0000220</td>
<td>SP</td>
<td></td>
<td>evsruw</td>
<td>Vector Shift Right Word Unsigned</td>
</tr>
<tr>
<td>EVX</td>
<td>0000221</td>
<td>SP</td>
<td></td>
<td>evsruw</td>
<td>Vector Shift Right Word Signed</td>
</tr>
<tr>
<td>EVX</td>
<td>0000222</td>
<td>SP</td>
<td></td>
<td>evsruwi0</td>
<td>Vector Shift Right Word Immediate Unsigned</td>
</tr>
<tr>
<td>EVX</td>
<td>0000223</td>
<td>SP</td>
<td></td>
<td>evsruwi0</td>
<td>Vector Shift Right Word Immediate Signed</td>
</tr>
<tr>
<td>EVX</td>
<td>0000224</td>
<td>SP</td>
<td></td>
<td>evlsw</td>
<td>Vector Shift Left Word</td>
</tr>
<tr>
<td>EVX</td>
<td>0000226</td>
<td>SP</td>
<td></td>
<td>evlswi0</td>
<td>Vector Shift Left Word Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>0000228</td>
<td>SP</td>
<td></td>
<td>evrl</td>
<td>Vector Rotate Left Word</td>
</tr>
<tr>
<td>EVX</td>
<td>0000229</td>
<td>SP</td>
<td></td>
<td>evsplat0</td>
<td>Vector Splat Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>000022A</td>
<td>SP</td>
<td></td>
<td>evrlwi0</td>
<td>Vector Rotate Left Word Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>000022B</td>
<td>SP</td>
<td></td>
<td>evsplat0</td>
<td>Vector Splat Fractional Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>000022C</td>
<td>SP</td>
<td></td>
<td>evmergehi</td>
<td>Vector Merge High</td>
</tr>
<tr>
<td>EVX</td>
<td>000022D</td>
<td>SP</td>
<td></td>
<td>evmergehi</td>
<td>Vector Merge Low</td>
</tr>
<tr>
<td>EVX</td>
<td>000022E</td>
<td>SP</td>
<td></td>
<td>evmergehil</td>
<td>Vector Merge High/Low</td>
</tr>
<tr>
<td>EVX</td>
<td>000022F</td>
<td>SP</td>
<td></td>
<td>evmergehol</td>
<td>Vector Merge Low/High</td>
</tr>
<tr>
<td>EVX</td>
<td>0000230</td>
<td>SP</td>
<td></td>
<td>evcmpgtu</td>
<td>Vector Compare Greater Than Unsigned</td>
</tr>
<tr>
<td>EVX</td>
<td>0000231</td>
<td>SP</td>
<td></td>
<td>evcmpgtu</td>
<td>Vector Compare Greater Than Signed</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep. Priv.</td>
<td>Cat$^{1}$</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>----------------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>10000232</td>
<td>SP</td>
<td>evcmppltu</td>
<td>Vector Compare Less Than Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000233</td>
<td>SP</td>
<td>evcmpplt</td>
<td>Vector Compare Less Than Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000234</td>
<td>SP</td>
<td>evcmpeq</td>
<td>Vector Compare Equal</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000240</td>
<td>V</td>
<td>vadduh</td>
<td>Vector Add Unsigned Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000242</td>
<td>V</td>
<td>vminuh</td>
<td>Vector Minimum Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000244</td>
<td>V</td>
<td>vsrh</td>
<td>Vector Shift Right Halfword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>10000246</td>
<td>V</td>
<td>vcmpgtuh[.]</td>
<td>Vector Compare Greater Than Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000248</td>
<td>V</td>
<td>vmlueuh</td>
<td>Vector Multiply Even Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000024A</td>
<td>V</td>
<td>vrfiz</td>
<td>Vector Round to Single-Precision Integer toward Zero</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000024C</td>
<td>V</td>
<td>vsplth</td>
<td>Vector Splat Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000024E</td>
<td>V</td>
<td>vupkhsh</td>
<td>Vector Unpack High Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>EVS</td>
<td>10000278</td>
<td>SP</td>
<td>evsel</td>
<td>Vector Select</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000280</td>
<td>SP.FV</td>
<td>evfadd</td>
<td>Vector Floating-Point Single-Precision Add</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000281</td>
<td>SP.FV</td>
<td>evfssub</td>
<td>Vector Floating-Point Single-Precision Subtract</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000282</td>
<td>SP</td>
<td>evfsmul</td>
<td>Vector Floating-Point Single-Precision Multiply</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000284</td>
<td>SP.FV</td>
<td>evfsabs</td>
<td>Vector Floating-Point Single-Precision Absolute Value</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000284</td>
<td>V</td>
<td>vsrwh</td>
<td>Vector Shift Right Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000285</td>
<td>SP.FV</td>
<td>evfnsabs</td>
<td>Vector Floating-Point Single-Precision Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000286</td>
<td>SP.FV</td>
<td>evfsneg</td>
<td>Vector Floating-Point Single-Precision Negate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000286</td>
<td>V</td>
<td>vcmpgtuw[.]</td>
<td>Vector Compare Greater Than Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000288</td>
<td>SP.FV</td>
<td>evfsmul</td>
<td>Vector Floating-Point Single-Precision Multiply</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000289</td>
<td>SP.FV</td>
<td>evfsdiv</td>
<td>Vector Floating-Point Single-Precision Divide</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000028A</td>
<td>V</td>
<td>vrfip</td>
<td>Vector Round to Single-Precision Integer toward +Infinity</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000028C</td>
<td>SP.FV</td>
<td>evfscmpgt</td>
<td>Vector Floating-Point Single-Precision Compare Greater Than</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000028C</td>
<td>V</td>
<td>vspltw</td>
<td>Vector Splat Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000028D</td>
<td>SP.FV</td>
<td>evfscmplt</td>
<td>Vector Floating-Point Single-Precision Compare Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000028E</td>
<td>SP.FV</td>
<td>evfscmpeq</td>
<td>Vector Floating-Point Single-Precision Compare Equal</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000028E</td>
<td>V</td>
<td>vupklb</td>
<td>Vector Unpack Low Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000290</td>
<td>SP.FV</td>
<td>evfsclui</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000291</td>
<td>SP.FV</td>
<td>evfscfsi</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000292</td>
<td>SP.FV</td>
<td>evfscfsf</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000293</td>
<td>SP.FV</td>
<td>evfscfsf</td>
<td>Vector Convert Floating-Point Single-Precision from Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000294</td>
<td>SP.FV</td>
<td>evfscctu</td>
<td>Vector Convert Floating-Point Single-Precision to Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000295</td>
<td>SP.FV</td>
<td>evfscctu</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000296</td>
<td>SP.FV</td>
<td>evfscctuf</td>
<td>Vector Convert Floating-Point Single-Precision to Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000297</td>
<td>SP.FV</td>
<td>evfscctuf</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000298</td>
<td>SP.FV</td>
<td>evfscelu</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000029A</td>
<td>SP.FV</td>
<td>evfscelsi</td>
<td>Vector Convert Floating-Point Single-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000029C</td>
<td>SP.FV</td>
<td>evfstsigt</td>
<td>Vector Floating-Point Single-Precision Test Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000029D</td>
<td>SP.FV</td>
<td>evfstslt</td>
<td>Vector Floating-Point Single-Precision Test Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000029E</td>
<td>SP.FV</td>
<td>evfsteq</td>
<td>Vector Floating-Point Single-Precision Test Equal</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C0</td>
<td>SP.FS</td>
<td>efsadd</td>
<td>Floating-Point Single-Precision Add</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C1</td>
<td>SP.FS</td>
<td>efsub</td>
<td>Floating-Point Single-Precision Subtract</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C4</td>
<td>SP.FS</td>
<td>efsabs</td>
<td>Floating-Point Single-Precision Absolute Value</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100002C4</td>
<td>V</td>
<td>vsr</td>
<td>Vector Shift Right</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002C5</td>
<td>SP.FS</td>
<td>efsnabs</td>
<td>Floating-Point Single-Precision Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep Priv</td>
<td>Cat</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>---------------</td>
<td>-----</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>100002C6</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsneg</td>
<td>Floating-Point Single-Precision Negate</td>
</tr>
<tr>
<td>VC</td>
<td>100002C6</td>
<td>V</td>
<td>V</td>
<td>vcmpgtfp[.]</td>
<td>Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>EVX</td>
<td>100002C8</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsmul</td>
<td>Floating-Point Single-Precision Multiply</td>
</tr>
<tr>
<td>EVX</td>
<td>100002C9</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efisdv</td>
<td>Floating-Point Single-Precision Divide</td>
</tr>
<tr>
<td>VX</td>
<td>100002CA</td>
<td>V</td>
<td>V</td>
<td>vrfim</td>
<td>Vector Round to Single-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>EVX</td>
<td>100002CC</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efscmpeqgt</td>
<td>Floating-Point Single-Precision Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>100002CD</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efscmpeqlt</td>
<td>Floating-Point Single-Precision Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>100002CE</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efscmpeqq</td>
<td>Floating-Point Single-Precision Compare Equal</td>
</tr>
<tr>
<td>VX</td>
<td>100002CE</td>
<td>V</td>
<td>V</td>
<td>vupklsh</td>
<td>Vector Unpack Low Signed Halfword</td>
</tr>
<tr>
<td>EVX</td>
<td>100002CF</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efsfcd</td>
<td>Floating-Point Single-Precision Convert from Double-Precision</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D0</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsfcui</td>
<td>Convert Floating-Point Single-Precision from Unsigned Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D1</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsfcsi</td>
<td>Convert Floating-Point Single-Precision from Signed Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D2</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsfcuf</td>
<td>Convert Floating-Point Single-Precision from Unsigned Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D3</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctsf</td>
<td>Convert Floating-Point Single-Precision from Signed Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D4</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctui</td>
<td>Convert Floating-Point Single-Precision to Unsigned Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D5</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctsi</td>
<td>Convert Floating-Point Single-Precision to Signed Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D6</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctuf</td>
<td>Convert Floating-Point Single-Precision to Unsigned Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D7</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctsf</td>
<td>Convert Floating-Point Single-Precision to Signed Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>100002D8</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctsz</td>
<td>Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>100002DA</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efsctsz</td>
<td>Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>100002DC</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efststgt</td>
<td>Floating-Point Single-Precision Test Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>100002DD</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efststlt</td>
<td>Floating-Point Single-Precision Test Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>100002DE</td>
<td>SP,FS</td>
<td>VFS</td>
<td>efststeq</td>
<td>Floating-Point Single-Precision Test Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E0</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efddadd</td>
<td>Floating-Point Double-Precision Add</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E1</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdsb</td>
<td>Floating-Point Double-Precision Subtract</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E2</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcuid</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E3</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcsid</td>
<td>Convert Floating-Point Double-Precision from Signed Integer Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E4</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfabs</td>
<td>Floating-Point Double-Precision Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E5</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfnabs</td>
<td>Floating-Point Double-Precision Negative Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E6</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfneg</td>
<td>Floating-Point Double-Precision Negate</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E8</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfmul</td>
<td>Floating-Point Double-Precision Multiply</td>
</tr>
<tr>
<td>EVX</td>
<td>100002E9</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfdiv</td>
<td>Floating-Point Double-Precision Divide</td>
</tr>
<tr>
<td>EVX</td>
<td>100002EA</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfctuidz</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>100002EB</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfctsidz</td>
<td>Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>100002ED</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcmpeqlt</td>
<td>Floating-Point Double-Precision Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>100002EC</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcmpeqgt</td>
<td>Floating-Point Double-Precision Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>100002EE</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcmpeqq</td>
<td>Floating-Point Double-Precision Compare Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>100002EF</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcs</td>
<td>Floating-Point Double-Precision Convert from Single-Precision</td>
</tr>
<tr>
<td>EVX</td>
<td>100002F0</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcfui</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>100002F1</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efdfcsf</td>
<td>Convert Floating-Point Double-Precision from Signed Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>100002F2</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efsfcuf</td>
<td>Convert Floating-Point Single-Precision from Unsigned Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>100002F3</td>
<td>SP,FD</td>
<td>VFS</td>
<td>efsfcf</td>
<td>Convert Floating-Point Double-Precision from Signed Fraction</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode</td>
<td>Cat</td>
<td>Mnemonic Instruction</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------</td>
<td>-----</td>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F4</td>
<td>SP.FD</td>
<td></td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F5</td>
<td>SP.FD</td>
<td></td>
<td>Convert Floating-Point Double-Precision to Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F6</td>
<td>SP.FD</td>
<td></td>
<td>Convert Floating-Point Double-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F7</td>
<td>SP.FD</td>
<td></td>
<td>Convert Floating-Point Double-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002F8</td>
<td>SP.FD</td>
<td></td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FA</td>
<td>SP.FD</td>
<td></td>
<td>Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FC</td>
<td>SP.FD</td>
<td></td>
<td>Floating-Point Double-Precision Test Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FD</td>
<td>SP.FD</td>
<td></td>
<td>Floating-Point Double-Precision Test Less Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100002FE</td>
<td>SP.FD</td>
<td></td>
<td>Floating-Point Double-Precision Test Equal</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000300</td>
<td>SP</td>
<td></td>
<td>Vector Load Double Word into Double Word Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000301</td>
<td>V</td>
<td></td>
<td>Vector Add Signed Byte Saturate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000302</td>
<td>SP</td>
<td></td>
<td>Vector Load Double Word into Double Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000303</td>
<td>SP</td>
<td></td>
<td>Vector Load Double Word into Two Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000304</td>
<td>SP</td>
<td></td>
<td>Vector Load Double Word into Four Halfwords Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000304</td>
<td>V</td>
<td></td>
<td>Vector Shift Right Algebraic Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000305</td>
<td>SP</td>
<td></td>
<td>Vector Load Double Word into Four Halfwords</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000306</td>
<td>V</td>
<td></td>
<td>Vector Compare Greater Than Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000308</td>
<td>SP</td>
<td></td>
<td>Vector Load Halfword into Halfwords Even and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000308</td>
<td>V</td>
<td></td>
<td>Vector Multiply Even Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000309</td>
<td>SP</td>
<td></td>
<td>Vector Load Halfword into Halfwords Even and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030A</td>
<td>V</td>
<td></td>
<td>Vector Convert From</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030C</td>
<td>SP</td>
<td></td>
<td>Vector Load Halfword into Halfword Odd Unsinged and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030C</td>
<td>V</td>
<td></td>
<td>Vector Splat Immediate Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030D</td>
<td>SP</td>
<td></td>
<td>Vector Load Halfword into Halfword Odd Unsinged and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030E</td>
<td>SP</td>
<td></td>
<td>Vector Load Halfword into Halfword Odd Signed and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030E</td>
<td>V</td>
<td></td>
<td>Vector Pack Pixel</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000030F</td>
<td>SP</td>
<td></td>
<td>Vector Load Halfword into Halfword Odd and Splat</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>10000310</td>
<td>SR</td>
<td></td>
<td>Multiply Low Halfword to Word Unsinged</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000311</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords Even</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000314</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords Odd Unsinged Indexed (zero-extended)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000315</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords Odd Unsinged (zero-extended)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000316</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000317</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords Odd Signed (with sign extension)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000318</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Word and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000318</td>
<td>SR</td>
<td></td>
<td>Multiply Accumulate Low Halfword to Word Modulo Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000319</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Word and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000031C</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000031D</td>
<td>SP</td>
<td></td>
<td>Vector Load Word into Two Halfwords and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000320</td>
<td>SP</td>
<td></td>
<td>Vector Store Doubleword of Doubleword Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000321</td>
<td>SP</td>
<td></td>
<td>Vector Store Double of Double</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000322</td>
<td>SP</td>
<td></td>
<td>Vector Store Double of Two Words Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000323</td>
<td>SP</td>
<td></td>
<td>Vector Store Double of Two Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000324</td>
<td>SP</td>
<td></td>
<td>Vector Store Double of Four Halfwords Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000325</td>
<td>SP</td>
<td></td>
<td>Vector Store Double of Four Halfwords</td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Mode</td>
<td>Dep.</td>
<td>Priv.</td>
<td>Cat 1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX 10000330</td>
<td>SP</td>
<td>evstwhex</td>
<td>Vector Store Word of Two Halfwords from Even Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000331</td>
<td>SP</td>
<td>evstwhe</td>
<td>Vector Store Word of Two Halfwords from Even</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000334</td>
<td>SP</td>
<td>evstwhox</td>
<td>Vector Store Word of Two Halfwords from Odd Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000338</td>
<td>SP</td>
<td>evstwmox</td>
<td>Vector Store Word of Two Halfwords from Odd Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 1000033d</td>
<td>SP</td>
<td>evstwho</td>
<td>Vector Store Word of Two Halfwords from Odd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 1000033E</td>
<td>SP</td>
<td>evstwwex</td>
<td>Vector Store Word of Word from Even Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 1000033F</td>
<td>SP</td>
<td>evstwwox</td>
<td>Vector Store Word of Word from Odd Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000340</td>
<td>SP</td>
<td>vaddshs</td>
<td>Vector Add Signed Halfword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000342</td>
<td>SP</td>
<td>vminsh</td>
<td>Vector Minimum Signed Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000344</td>
<td>SP</td>
<td>vsrah</td>
<td>Vector Shift Right Algebraic Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000346</td>
<td>SP</td>
<td>vcmptghs</td>
<td>Vector Compare Greater Than Signed Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 10000348</td>
<td>SP</td>
<td>vmulesh</td>
<td>Vector Multiply Even Signed Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 1000034A</td>
<td>SP</td>
<td>vcfsx</td>
<td>Vector Convert From Signed Fixed-Point Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 1000034C</td>
<td>SP</td>
<td>vsplish</td>
<td>Vector Splat Immediate Signed Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 1000034E</td>
<td>SP</td>
<td>vupkhpx</td>
<td>Vector Unpack High Pixel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000350</td>
<td>SR</td>
<td>lma</td>
<td>Multiply Low Halfword to Word Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000355</td>
<td>SR</td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000380</td>
<td>V</td>
<td>vaddshs</td>
<td>Vector Add Signed Halfword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000382</td>
<td>V</td>
<td>vminsw</td>
<td>Vector Minimum Signed Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000384</td>
<td>V</td>
<td>vsraw</td>
<td>Vector Shift Right Algebraic Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000386</td>
<td>V</td>
<td>vcmptgshs</td>
<td>Vector Compare Greater Than Signed Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000038A</td>
<td>V</td>
<td>vctuxs</td>
<td>Vector Convert To Signed Fixed-Point Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000038C</td>
<td>V</td>
<td>vsplitw</td>
<td>Vector Splat Immediate Signed Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000398</td>
<td>SR</td>
<td>LMA</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 100003CA</td>
<td>V</td>
<td>vcmpbfps</td>
<td>Vector Compare Bounds Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 100003CE</td>
<td>V</td>
<td>vctxsx</td>
<td>Vector Convert To Signed Fixed-Point Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 100003DC</td>
<td>SR</td>
<td>lma</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000400</td>
<td>V</td>
<td>vsububm</td>
<td>Vector Subtract Unsigned Byte Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000402</td>
<td>V</td>
<td>vagub</td>
<td>Vector Average Unsigned Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000403</td>
<td>SP</td>
<td>evmhesf</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000404</td>
<td>V</td>
<td>vand</td>
<td>Vector Logical AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000407</td>
<td>SP</td>
<td>evmhosf</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000408</td>
<td>SP</td>
<td>evmheumi</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000409</td>
<td>SP</td>
<td>evmhesmi</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000040A</td>
<td>V</td>
<td>vmmaxfp</td>
<td>Vector Maximum Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000040B</td>
<td>SP</td>
<td>evmhesmf</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000040C</td>
<td>SP</td>
<td>evmhoumi</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000040C</td>
<td>V</td>
<td>vslo</td>
<td>Vector Shift Left by Octet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000040D</td>
<td>SP</td>
<td>evmhosmi</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 1000040F</td>
<td>SP</td>
<td>evmhossf</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000423</td>
<td>SP</td>
<td>evmhesfa</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX 10000427</td>
<td>SP</td>
<td>evmhossfa</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode/Dep/Priv</td>
<td>Cat&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>---------------</td>
<td>----------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>10000428</td>
<td>SP</td>
<td>evmheumia</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000429</td>
<td>SP</td>
<td>evmhesmia</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000042B</td>
<td>SP</td>
<td>evmhesmfa</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000042C</td>
<td>SP</td>
<td>evmhounia</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000042D</td>
<td>SP</td>
<td>evmhosmia</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000042F</td>
<td>SP</td>
<td>evmhosmfa</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000440</td>
<td>V</td>
<td>vsubuhm</td>
<td>Vector Subtract Unsigned Halfword Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000442</td>
<td>V</td>
<td>vavguh</td>
<td>Vector Average Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000444</td>
<td>V</td>
<td>vandc</td>
<td>Vector Logical AND with Complement</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000447</td>
<td>SP</td>
<td>evmwhssf</td>
<td>Vector Multiply Word High Signed, Saturate, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000448</td>
<td>SP</td>
<td>evmwlumi</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000449</td>
<td>V</td>
<td>vminfp</td>
<td>Vector Minimum Single-Precision</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000044A</td>
<td>SP</td>
<td>evmhwumi</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>1000044C</td>
<td>V</td>
<td>vsro</td>
<td>Vector Shift Right by Octet</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000044D</td>
<td>SP</td>
<td>evmwhsmi</td>
<td>Vector Multiply Word High Signed, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000044E</td>
<td>SP</td>
<td>evmwhsmsf</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000453</td>
<td>SP</td>
<td>evmwsasf</td>
<td>Vector Multiply Word Signed, Saturate, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000458</td>
<td>SP</td>
<td>evmwsumi</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000459</td>
<td>SP</td>
<td>evmwsmsi</td>
<td>Vector Multiply Word Signed, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000467</td>
<td>SP</td>
<td>evmwhssfa</td>
<td>Vector Multiply Word High Signed, Saturate, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000468</td>
<td>SP</td>
<td>evmwlumia</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000046C</td>
<td>SP</td>
<td>evmwhumia</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000046D</td>
<td>SP</td>
<td>evmwhsmsia</td>
<td>Vector Multiply Word High Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000046F</td>
<td>SP</td>
<td>evmwhsmsfa</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000473</td>
<td>SP</td>
<td>evmwsssa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000478</td>
<td>SP</td>
<td>evmwumia</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>10000479</td>
<td>SP</td>
<td>evmwsmsia</td>
<td>Vector Multiply Word Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>1000047B</td>
<td>SP</td>
<td>evmwsmsfa</td>
<td>Vector Multiply Word Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000480</td>
<td>V</td>
<td>vsubuwsm</td>
<td>Vector Subtract Unsigned Word Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000482</td>
<td>V</td>
<td>vavguw</td>
<td>Vector Average Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000484</td>
<td>V</td>
<td>vor</td>
<td>Vector Logical OR</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C0</td>
<td>SP</td>
<td>evaddusiaaw</td>
<td>Vector Add Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C1</td>
<td>SP</td>
<td>evaddssiaaw</td>
<td>Vector Add Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C2</td>
<td>SP</td>
<td>evsubusiaaw</td>
<td>Vector Subtract Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C3</td>
<td>SP</td>
<td>evsubssiaaw</td>
<td>Vector Subtract Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C4</td>
<td>SP</td>
<td>evmra</td>
<td>Initialize Accumulator</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>100004C4</td>
<td>V</td>
<td>vxor</td>
<td>Vector Logical XOR</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C6</td>
<td>SP</td>
<td>evdiwsw</td>
<td>Vector Divide Word Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C7</td>
<td>SP</td>
<td>evdiwuw</td>
<td>Vector Divide Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>100004C8</td>
<td>SP</td>
<td>evaddumiaaw</td>
<td>Vector Add Unsigned, Modulo, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep.</td>
<td>Priv</td>
<td>Cat</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>-----------</td>
<td>------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>100004C9</td>
<td>SP</td>
<td></td>
<td></td>
<td>evaddsmiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>100004CA</td>
<td>SP</td>
<td></td>
<td></td>
<td>evsubfumiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>100004CB</td>
<td>SP</td>
<td></td>
<td></td>
<td>evsubfsmiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000500</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmheusiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000501</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhesiaaw</td>
</tr>
<tr>
<td>VX</td>
<td>10000502</td>
<td>V</td>
<td></td>
<td></td>
<td>vavggb</td>
</tr>
<tr>
<td>EVX</td>
<td>10000503</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhessfaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000504</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhouusiaaw</td>
</tr>
<tr>
<td>VX</td>
<td>10000504</td>
<td>V</td>
<td></td>
<td></td>
<td>vnor</td>
</tr>
<tr>
<td>EVX</td>
<td>10000505</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000507</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhossfaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000508</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmheumiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000509</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhesniaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050B</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhesmfiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhoumiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhosniaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050F</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhosmfiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000528</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhegumiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000529</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhegismaa</td>
</tr>
<tr>
<td>EVX</td>
<td>1000052B</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhegsmiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>1000052C</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhogumiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>1000052D</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmhogsmiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>1000052F</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlusiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000540</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwssluaaw</td>
</tr>
<tr>
<td>VX</td>
<td>10000542</td>
<td>V</td>
<td></td>
<td></td>
<td>vavgsh</td>
</tr>
<tr>
<td>EVX</td>
<td>10000548</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwlimiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000549</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwismaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000553</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwssfaa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000558</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwumiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000559</td>
<td>SP</td>
<td></td>
<td></td>
<td>evmwsmiaa</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep.¹ Priv¹</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>-----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>0000055B</td>
<td>SP</td>
<td></td>
<td>evmwsfmw</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>00000580</td>
<td>SP</td>
<td></td>
<td>evmheusianw</td>
<td>Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>VX</td>
<td>00000580</td>
<td>V</td>
<td></td>
<td>vsubcuw</td>
<td>Vector Subtract and Write Carry-Out Unsigned Word</td>
</tr>
<tr>
<td>EVX</td>
<td>00000581</td>
<td>SP</td>
<td></td>
<td>evmheussianw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>VX</td>
<td>00000582</td>
<td>V</td>
<td></td>
<td>vavgsw</td>
<td>Vector Average Signed Word</td>
</tr>
<tr>
<td>EVX</td>
<td>00000583</td>
<td>SP</td>
<td></td>
<td>evmheussianw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>00000584</td>
<td>SP</td>
<td></td>
<td>evmhosianw</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>00000585</td>
<td>SP</td>
<td></td>
<td>evmhosianw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>00000587</td>
<td>SP</td>
<td></td>
<td>evmheuminw</td>
<td>Vector Multiply Halfwords, Even, Unsaturated, Saturate, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>00000588</td>
<td>SP</td>
<td></td>
<td>evmheuminw</td>
<td>Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>00000589</td>
<td>SP</td>
<td></td>
<td>evmheuminw</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>0000058B</td>
<td>SP</td>
<td></td>
<td>evmheuminw</td>
<td>Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>0000058C</td>
<td>SP</td>
<td></td>
<td>evmhoumianw</td>
<td>Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>0000058D</td>
<td>SP</td>
<td></td>
<td>evmhosmianw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>0000058F</td>
<td>SP</td>
<td></td>
<td>evmhosmianw</td>
<td>Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>000005A8</td>
<td>SP</td>
<td></td>
<td>evmheugumian</td>
<td>Vector Multiply Halfwords, Even, Guarded, Unsigned, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005A9</td>
<td>SP</td>
<td></td>
<td>evmhegsian</td>
<td>Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005AB</td>
<td>SP</td>
<td></td>
<td>evmhegsfan</td>
<td>Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005AC</td>
<td>SP</td>
<td></td>
<td>evmhogumian</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005AD</td>
<td>SP</td>
<td></td>
<td>evmhogsian</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005AF</td>
<td>SP</td>
<td></td>
<td>evmhogsmfan</td>
<td>Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005C0</td>
<td>SP</td>
<td></td>
<td>evmwusianw</td>
<td>Vector Multiply Word Low Unsaturated, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>000005C1</td>
<td>SP</td>
<td></td>
<td>evmwlssianw</td>
<td>Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>000005C8</td>
<td>SP</td>
<td></td>
<td>evmwulfianw</td>
<td>Vector Multiply Word Low Unsaturated, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>000005C9</td>
<td>SP</td>
<td></td>
<td>evmwlsianw</td>
<td>Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>000005D3</td>
<td>SP</td>
<td></td>
<td>evmwssfan</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005D8</td>
<td>SP</td>
<td></td>
<td>evmwumian</td>
<td>Vector Multiply Word Unsaturated, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005D9</td>
<td>SP</td>
<td></td>
<td>evmwsian</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>000005DB</td>
<td>SP</td>
<td></td>
<td>evmwsmfan</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>VX</td>
<td>00000600</td>
<td>V</td>
<td></td>
<td>vsububs</td>
<td>Vector Subtract Unsigned Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>00000604</td>
<td>V</td>
<td></td>
<td>mfvscr</td>
<td>Move From VSCR</td>
</tr>
<tr>
<td>VX</td>
<td>00000608</td>
<td>V</td>
<td></td>
<td>vsuml4ubs</td>
<td>Vector Sum across Quarter Unsigned Byte Saturate</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep.¹</td>
<td>Priv¹</td>
<td>Cat¹</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------------</td>
<td>-------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>VX</td>
<td>10000640</td>
<td>V</td>
<td></td>
<td></td>
<td>vsubuhs</td>
</tr>
<tr>
<td>VX</td>
<td>10000644</td>
<td>V</td>
<td></td>
<td></td>
<td>mtvscr</td>
</tr>
<tr>
<td>VX</td>
<td>10000648</td>
<td>V</td>
<td></td>
<td></td>
<td>vsnum4hs</td>
</tr>
<tr>
<td>VX</td>
<td>10000680</td>
<td>V</td>
<td></td>
<td></td>
<td>vsnum4shs</td>
</tr>
<tr>
<td>VX</td>
<td>10000688</td>
<td>V</td>
<td></td>
<td></td>
<td>vsubw</td>
</tr>
<tr>
<td>VX</td>
<td>10000700</td>
<td>V</td>
<td></td>
<td></td>
<td>vsnum2w</td>
</tr>
<tr>
<td>VX</td>
<td>10000708</td>
<td>V</td>
<td></td>
<td></td>
<td>vsubsbs</td>
</tr>
<tr>
<td>VX</td>
<td>10000740</td>
<td>V</td>
<td></td>
<td></td>
<td>vsnum2s</td>
</tr>
<tr>
<td>VX</td>
<td>10000780</td>
<td>V</td>
<td></td>
<td></td>
<td>vsbsh</td>
</tr>
<tr>
<td>VX</td>
<td>10000788</td>
<td>V</td>
<td></td>
<td></td>
<td>vsnumw</td>
</tr>
<tr>
<td>D8</td>
<td>18000000</td>
<td>VLE e_lbzu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000100</td>
<td>VLE e_lhzu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000200</td>
<td>VLE e_lwzu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000300</td>
<td>VLE e_lhau</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000400</td>
<td>VLE e_stbu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000500</td>
<td>VLE e_sthu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000600</td>
<td>VLE e_stwu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000800</td>
<td>VLE e_lmw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>18000900</td>
<td>VLE e_stmw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCI8</td>
<td>18080000 SR</td>
<td>VLE e_add[.]</td>
<td></td>
<td></td>
<td>Add Scaled Immediate</td>
</tr>
<tr>
<td>SCI8</td>
<td>18090000 SR</td>
<td>VLE e_addic[.]</td>
<td></td>
<td></td>
<td>Add Scaled Immediate Carrying</td>
</tr>
<tr>
<td>SCI8</td>
<td>180A0000 SR</td>
<td>VLE e_mul[.]</td>
<td></td>
<td></td>
<td>Multiply Low Scaled Immediate</td>
</tr>
<tr>
<td>SCI8</td>
<td>180A8000 SR</td>
<td>VLE e_cmpi</td>
<td></td>
<td></td>
<td>Compare Scaled Immediate Word</td>
</tr>
<tr>
<td>SCI8</td>
<td>180B0000 SR</td>
<td>VLE e_cmpi</td>
<td></td>
<td></td>
<td>Subtract From Scaled Immediate Carrying</td>
</tr>
<tr>
<td>SCI8</td>
<td>180C0000 SR</td>
<td>VLE e_andi[.]</td>
<td></td>
<td></td>
<td>AND Scaled Immediate</td>
</tr>
<tr>
<td>SCI8</td>
<td>180D0000 SR</td>
<td>VLE e_or[.]</td>
<td></td>
<td></td>
<td>OR Scaled Immediate</td>
</tr>
<tr>
<td>SCI8</td>
<td>180E0000 SR</td>
<td>VLE e_xor[.]</td>
<td></td>
<td></td>
<td>XOR Scaled Immediate</td>
</tr>
<tr>
<td>SCI8</td>
<td>1808A800 SR</td>
<td>VLE e_cmpli</td>
<td></td>
<td></td>
<td>Compare Logical Scaled Immediate Word</td>
</tr>
<tr>
<td>D</td>
<td>1C000000</td>
<td>VLE e_add16i</td>
<td></td>
<td></td>
<td>Add Immediate</td>
</tr>
<tr>
<td>OIM5</td>
<td>2000-----</td>
<td>VLE se_add</td>
<td></td>
<td></td>
<td>Add Immediate Short Form</td>
</tr>
<tr>
<td>OIM5</td>
<td>2200-----</td>
<td>VLE se_cmpli</td>
<td></td>
<td></td>
<td>Compare Logical Immediate Word</td>
</tr>
<tr>
<td>OIM5</td>
<td>2400-----</td>
<td>VLE se_sub[.]</td>
<td></td>
<td></td>
<td>Subtract Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>2A00-----</td>
<td>VLE se_cmpi</td>
<td></td>
<td></td>
<td>Compare Immediate Word Short Form</td>
</tr>
<tr>
<td>IM5</td>
<td>2C00-----</td>
<td>VLE se_bmaski</td>
<td></td>
<td></td>
<td>Bit Mask Generate Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>2E00-----</td>
<td>VLE se_andi</td>
<td></td>
<td></td>
<td>AND Immediate Short Form</td>
</tr>
<tr>
<td>D</td>
<td>30000000</td>
<td>VLE e_lbz</td>
<td></td>
<td></td>
<td>Load Byte and Zero</td>
</tr>
<tr>
<td>D</td>
<td>34000000</td>
<td>VLE e_stb</td>
<td></td>
<td></td>
<td>Store Byte</td>
</tr>
<tr>
<td>D</td>
<td>38000000</td>
<td>VLE e_lha</td>
<td></td>
<td></td>
<td>Load Halfword Algebraic</td>
</tr>
<tr>
<td>RR</td>
<td>4000-----</td>
<td>VLE e_sw</td>
<td></td>
<td></td>
<td>Shift Right Word</td>
</tr>
<tr>
<td>RR</td>
<td>4100-----</td>
<td>VLE e_saw</td>
<td></td>
<td></td>
<td>Shift Right Algebraic Word</td>
</tr>
<tr>
<td>RR</td>
<td>4200-----</td>
<td>VLE e_slw</td>
<td></td>
<td></td>
<td>Shift Left Word</td>
</tr>
<tr>
<td>RR</td>
<td>4400-----</td>
<td>VLE e_or</td>
<td></td>
<td></td>
<td>OR Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>4500-----</td>
<td>VLE se_andc</td>
<td></td>
<td></td>
<td>AND with Complement Short Form</td>
</tr>
<tr>
<td>RR</td>
<td>4600-----</td>
<td>VLE se_and</td>
<td></td>
<td></td>
<td>AND Short Form</td>
</tr>
<tr>
<td>IM7</td>
<td>4800-----</td>
<td>VLE se_li</td>
<td></td>
<td></td>
<td>Load Immediate Short Form</td>
</tr>
<tr>
<td>D</td>
<td>50000000</td>
<td>VLE e_lwz</td>
<td></td>
<td></td>
<td>Load Word and Zero</td>
</tr>
<tr>
<td>D</td>
<td>54000000</td>
<td>VLE e_stw</td>
<td></td>
<td></td>
<td>Store Word</td>
</tr>
<tr>
<td>D</td>
<td>58000000</td>
<td>VLE e_lhz</td>
<td></td>
<td></td>
<td>Load Halfword and Zero</td>
</tr>
<tr>
<td>D</td>
<td>5C000000</td>
<td>VLE e_sht</td>
<td></td>
<td></td>
<td>Store Halfword</td>
</tr>
<tr>
<td>IM5</td>
<td>6000-----</td>
<td>VLE se_bcr</td>
<td></td>
<td></td>
<td>Bit Clear Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6200-----</td>
<td>VLE se_bgeni</td>
<td></td>
<td></td>
<td>Bit Generate Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6400-----</td>
<td>VLE se_bsct</td>
<td></td>
<td></td>
<td>Bit Set Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6600-----</td>
<td>VLE se_bts</td>
<td></td>
<td></td>
<td>Bit Test Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6800-----</td>
<td>VLE se_srw</td>
<td></td>
<td></td>
<td>Shift Right Word Immediate Short Form</td>
</tr>
<tr>
<td>IM5</td>
<td>6A00-----</td>
<td>VLE se_sraw</td>
<td></td>
<td></td>
<td>Shift Right Algebraic Word Immediate</td>
</tr>
<tr>
<td>IM5</td>
<td>6C00-----</td>
<td>VLE se_slw</td>
<td></td>
<td></td>
<td>Shift Left Word Immediate Short Form</td>
</tr>
<tr>
<td>LI20</td>
<td>70000000 SR</td>
<td>VLE e_li</td>
<td></td>
<td></td>
<td>Load Immediate</td>
</tr>
<tr>
<td>I16A</td>
<td>70008800</td>
<td>VLE e_add2i.</td>
<td></td>
<td></td>
<td>Add (2 operand) Immediate and Record</td>
</tr>
<tr>
<td>I16A</td>
<td>70009000</td>
<td>VLE e_add2is</td>
<td></td>
<td></td>
<td>Add (2 operand) Immediate Shifted</td>
</tr>
<tr>
<td>I16A</td>
<td>70009800</td>
<td>VLE e_cmp16i</td>
<td></td>
<td></td>
<td>Compare Immediate Word</td>
</tr>
<tr>
<td>I16A</td>
<td>7000A000</td>
<td>VLE e_mul2i</td>
<td></td>
<td></td>
<td>Multiply (2 operand) Low Immediate</td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)2</td>
<td>Mode Dep. Priv</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>I16A</td>
<td>7000A800</td>
<td>VLE</td>
<td>e_cmpl16i</td>
<td>Compare Logical Immediate Word</td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>7000B000</td>
<td>VLE</td>
<td>e_cmph16i</td>
<td>Compare Halfword Immediate</td>
<td></td>
</tr>
<tr>
<td>I16A</td>
<td>7000B800</td>
<td>VLE</td>
<td>e_cmphl16i</td>
<td>Compare Halfword Logical Immediate</td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000C800</td>
<td>SR VLE</td>
<td>e_or2i</td>
<td>OR (two operand) Immediate</td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000D000</td>
<td>VLE</td>
<td>e_or2is</td>
<td>OR (2 operand) Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000E000</td>
<td>SR VLE</td>
<td>e_and2i</td>
<td>AND (two operand) Immediate</td>
<td></td>
</tr>
<tr>
<td>I16L</td>
<td>7000E800</td>
<td>VLE</td>
<td>e_and2is</td>
<td>AND (2 operand) Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>74000000</td>
<td>VLE</td>
<td>e_rlwimi</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>74000001</td>
<td>VLE</td>
<td>e_rlwinm</td>
<td>Rotate Left Word Immediate then AND with Mask</td>
<td></td>
</tr>
<tr>
<td>BD24</td>
<td>78000000</td>
<td>VLE e_b[]</td>
<td></td>
<td>Branch [and Link]</td>
<td></td>
</tr>
<tr>
<td>BD15</td>
<td>7A000000</td>
<td>CT VLE</td>
<td>e_bc[]</td>
<td>Branch Conditional [and Link]</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000000</td>
<td>B</td>
<td>cmp</td>
<td>Compare</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000008</td>
<td>B</td>
<td>tw</td>
<td>Trap Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000C</td>
<td>V</td>
<td>lvsl</td>
<td>Load Vector for Shift Left Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000E</td>
<td>V</td>
<td>lvbx</td>
<td>Load Vector Element Byte Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000010</td>
<td>SR B</td>
<td>subf[]</td>
<td>Subtract From Carrying</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000012</td>
<td>SR 64</td>
<td>mulhd[]</td>
<td>Multiply High Doubleword Unsigned</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000014</td>
<td>SR B</td>
<td>addc[]</td>
<td>Add Carrying</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000016</td>
<td>SR B</td>
<td>mulhw[]</td>
<td>Multiply High Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00001C</td>
<td>VLE e_cmph</td>
<td></td>
<td>Compare Halfword</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>7C00001E</td>
<td>B</td>
<td>isel</td>
<td>Integer Select</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000020</td>
<td>VLE e_mcrf</td>
<td></td>
<td>Move CR Field</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000026</td>
<td>B</td>
<td>mfcrc</td>
<td>Move From Condition Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000028</td>
<td>B</td>
<td>lwax</td>
<td>Load Word And Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002A</td>
<td>64</td>
<td>ldx</td>
<td>Load Doubleword Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002C</td>
<td>B</td>
<td>icbt</td>
<td>Instruction Cache Block Touch</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>B</td>
<td>lwzx</td>
<td>Load Word and Zero Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000030</td>
<td>SR B</td>
<td>siw[.]</td>
<td>Shift Left Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000034</td>
<td>SR B</td>
<td>cntlz[w[.]</td>
<td>Count Leading Zeros Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000036</td>
<td>SR B</td>
<td>sid[.]</td>
<td>Shift Left Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000038</td>
<td>SR B</td>
<td>and[.]</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00003A</td>
<td>P E.PD:64</td>
<td>idepx</td>
<td>Load Doubleword by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00003E</td>
<td>P E.PD</td>
<td>lwepx</td>
<td>Load Word by External Process Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000040</td>
<td>B</td>
<td>cmpol</td>
<td>Compare Logical</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000042</td>
<td>VLE e_error</td>
<td></td>
<td>Condition Register NOR</td>
<td></td>
</tr>
<tr>
<td>ESC</td>
<td>7C000048</td>
<td>VLE e_sc</td>
<td></td>
<td>System Call</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00004C</td>
<td>V</td>
<td>lvslr</td>
<td>Load Vector for Shift Right Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>V</td>
<td>lvbxh</td>
<td>Load Vector Element Halfword Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000050</td>
<td>SR B</td>
<td>subf[[]]</td>
<td>Subtract From</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00005C</td>
<td>VLE e_cmplh</td>
<td></td>
<td>Compare Halfword Logical</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000068</td>
<td>B</td>
<td>lbaxr</td>
<td>Load Byte and Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006A</td>
<td>64</td>
<td>ldux</td>
<td>Load Doubleword with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006C</td>
<td>B</td>
<td>dbcst</td>
<td>Data Cache Block Store</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006E</td>
<td>B</td>
<td>lwux</td>
<td>Load Word and Zero with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000070</td>
<td>SR VLE</td>
<td>e_slw[.]</td>
<td>Shift Left Word Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000074</td>
<td>SR 64</td>
<td>cntlzd[.]</td>
<td>Count Leading Zeros Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000078</td>
<td>SR B</td>
<td>andc[.]</td>
<td>AND with Complement</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00007C</td>
<td>WT wait</td>
<td></td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00007E</td>
<td>E.PD</td>
<td>dbcstep</td>
<td>Data Cache Block Store by External PID</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000088</td>
<td>64</td>
<td>td</td>
<td>Trap Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00008E</td>
<td>V</td>
<td>lvwxx</td>
<td>Load Vector Element Word Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000092</td>
<td>SR 64</td>
<td>mulhd[.]</td>
<td>Multiply High Doubleword</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000096</td>
<td>SR B</td>
<td>mulhw[.]</td>
<td>Multiply High Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000A6</td>
<td>P B</td>
<td>mfcmsr</td>
<td>Move From Machine State Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000A8</td>
<td>64</td>
<td>ldaxr</td>
<td>Load Doubleword And Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000AC</td>
<td>B</td>
<td>dbcf</td>
<td>Data Cache Block Flush</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000AE</td>
<td>B</td>
<td>lbzx</td>
<td>Load Byte and Zero Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000BE</td>
<td>P E.PD</td>
<td>lbepx</td>
<td>Load Byte by External Process ID Indexed</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)2</td>
<td>Mode (dep. priv)1</td>
<td>Cat1</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>------------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>X</td>
<td>7C00000C</td>
<td>V</td>
<td>lvx</td>
<td>Load Vector Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C00000D</td>
<td>B nego</td>
<td>Negate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000E</td>
<td>B lharx</td>
<td>Load Halfword and Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000F</td>
<td>B lbzux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000F</td>
<td>B popcntb</td>
<td>Population Count Bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000E</td>
<td>B nor</td>
<td>NOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00000F</td>
<td>P E.PD dcbfep</td>
<td>Data Cache Block Flush by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000012</td>
<td>VLE e_crandc</td>
<td>Condition Register AND with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>7C000016</td>
<td>E wrtee</td>
<td>Write MSR External Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00001C</td>
<td>M ECL dcbtstls</td>
<td>Data Cache Block Touch for Store and Lock Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00001E</td>
<td>V stvebx</td>
<td>Store Vector Element Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C00001D</td>
<td>B subfeo</td>
<td>Subtract From Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C00001E</td>
<td>B addeo</td>
<td>Add Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C000020</td>
<td>P E mtcfr</td>
<td>Move To Condition Register Fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000024</td>
<td>X stdx</td>
<td>Store Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002D</td>
<td>B stwx</td>
<td>Store Word Conditional Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>B stw</td>
<td>Store Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00003A</td>
<td>P E.PD;64 stdexpx</td>
<td>Store Doubleword by External Process ID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00003E</td>
<td>P E.PD stwpx</td>
<td>Store Word by External Process ID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000046</td>
<td>P E wrteei</td>
<td>Write MSR External Enable Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00004C</td>
<td>M ECL dcbtls</td>
<td>Data Cache Block Touch and Lock Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>V stvehx</td>
<td>Store Vector Element Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006A</td>
<td>64 stdtx</td>
<td>Store Doubleword with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00006E</td>
<td>B stwux</td>
<td>Store Word with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000082</td>
<td>VLE e_crxor</td>
<td>Condition Register XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00008D</td>
<td>M ECL icblq</td>
<td>Instruction Cache Block Lock Query</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000090</td>
<td>B stvewx</td>
<td>Store Vector Element Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000094</td>
<td>B add[ . ]o</td>
<td>Add to Zero Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00009C</td>
<td>H.E.PC msgsnd</td>
<td>Message Send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000108</td>
<td>64 stdcx</td>
<td>Store Doubleword Conditional Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00010E</td>
<td>B stb</td>
<td>Store Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000113</td>
<td>P E.PD stbepx</td>
<td>Store Byte by External Process ID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000124</td>
<td>VLE e_crmnd</td>
<td>Condition Register NAND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000138</td>
<td>M ECL icblc</td>
<td>Instruction Cache Block Lock Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00014C</td>
<td>V stvx</td>
<td>Store Vector Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C00015A</td>
<td>B subfmeo</td>
<td>Subtract From Minus One Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000160</td>
<td>64 mul[ . ]o</td>
<td>Multiply Low Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C00016E</td>
<td>B addmeo</td>
<td>Add to Minus One Extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000177</td>
<td>64 mulw[ . ]o</td>
<td>Multiply Low Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000190</td>
<td>H.E.PC msgclr</td>
<td>Message Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000194</td>
<td>B dcbtst</td>
<td>Data Cache Block Touch for Store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00019E</td>
<td>B stbux</td>
<td>Store Byte with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00019E</td>
<td>P E.PD dcbtstep</td>
<td>Data Cache Block Touch for Store by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000202</td>
<td>VLE e_crand</td>
<td>Condition Register AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000214</td>
<td>P E.DC mfcdcrx</td>
<td>Move From Device Control Register Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000214</td>
<td>P E.PD lvepxl</td>
<td>Load Vector by External Process ID Indexed LRU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00021A</td>
<td>B add[ . ]o</td>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>7C000221</td>
<td>E.HV ehpri</td>
<td>Embedded Hypervisor Privilege</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000222</td>
<td>B dcbt</td>
<td>Data Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00022E</td>
<td>B lhzx</td>
<td>Load Halfword and Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000230</td>
<td>VLE e_rlw[ . ]</td>
<td>Rotate Left Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000238</td>
<td>B eqv[ . ]</td>
<td>Equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00023E</td>
<td>P E.PD lhexp</td>
<td>Load Halfword by External Process ID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000242</td>
<td>VLE e_crevq</td>
<td>Condition Register Equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000246</td>
<td>E.DC mfcdcrxr</td>
<td>Move From Device Control Register User-mode Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00024E</td>
<td>P E.PD lvepx</td>
<td>Load Vector by External Process ID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00026E</td>
<td>B lhzx</td>
<td>Load Halfword and Zero with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000270</td>
<td>VLE e_rlw[ . ]</td>
<td>Rotate Left Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000278</td>
<td>B xor[ . ]</td>
<td>XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)</td>
<td>Mode Dep. Priv.</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>X</td>
<td>7C00027E</td>
<td>P</td>
<td>E.PD</td>
<td>dcbtep</td>
<td>Data Cache Block Touch by External PID</td>
</tr>
<tr>
<td>XFX</td>
<td>7C000286</td>
<td>P</td>
<td>E.DC</td>
<td>mfmdcr</td>
<td>Move From Device Control Register</td>
</tr>
<tr>
<td>X</td>
<td>7C00028C</td>
<td>P</td>
<td>E.CD</td>
<td>dcread</td>
<td>Data Cache Read</td>
</tr>
<tr>
<td>XFX</td>
<td>7C00029C</td>
<td>O</td>
<td>E.PM</td>
<td>mtpmr</td>
<td>Move From Performance Monitor Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C0002A6</td>
<td>O</td>
<td>B</td>
<td>mspsr</td>
<td>Move From Special Purpose Register</td>
</tr>
<tr>
<td>X</td>
<td>7C0002CA</td>
<td>64</td>
<td>B</td>
<td>iwait</td>
<td>Load Word Algebraic Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C0002EA</td>
<td>64</td>
<td>B</td>
<td>ihaux</td>
<td>Load Word Algebraic with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C0002EE</td>
<td>64</td>
<td>B</td>
<td>ihaux</td>
<td>Load Halfword Algebraic with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000306</td>
<td>P</td>
<td>E.DC</td>
<td>mtmdcr</td>
<td>Move To Device Control Register Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00030C</td>
<td>M</td>
<td>ECL</td>
<td>dcbie</td>
<td>Data Cache Block Lock Clear</td>
</tr>
<tr>
<td>X</td>
<td>7C00032E</td>
<td>B</td>
<td>sthx</td>
<td>Store Halfword Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00033E</td>
<td>SR</td>
<td>B</td>
<td>or[,]</td>
<td>OR with Complement</td>
</tr>
<tr>
<td>X</td>
<td>7C00033E</td>
<td>SR</td>
<td>P</td>
<td>sthepx</td>
<td>Store Halfword by External Process ID Indexed</td>
</tr>
<tr>
<td>XL</td>
<td>7C000342</td>
<td>VLE</td>
<td>e_crocr</td>
<td>Condition Register OR with Complement</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000346</td>
<td>M</td>
<td>E.DC</td>
<td>mtdcru</td>
<td>Move To Device Control Register User-mode Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00034D</td>
<td>M</td>
<td>ECL</td>
<td>dcbiql</td>
<td>Data Cache Block Lock Query</td>
</tr>
<tr>
<td>X</td>
<td>7C00035E</td>
<td>B</td>
<td>sthux</td>
<td>Store Halfword with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000378</td>
<td>SR</td>
<td>B</td>
<td>or[,]</td>
<td>OR</td>
</tr>
<tr>
<td>XL</td>
<td>7C000382</td>
<td>VLE</td>
<td>e_cror</td>
<td>Condition Register OR</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C000386</td>
<td>P</td>
<td>E.DC</td>
<td>mtmdcr</td>
<td>Move To Device Control Register</td>
</tr>
<tr>
<td>XO</td>
<td>7C00038C</td>
<td>H</td>
<td>E.CI</td>
<td>dci</td>
<td>Data Cache Invalidate</td>
</tr>
<tr>
<td>XO</td>
<td>7C000392</td>
<td>SR</td>
<td>64</td>
<td>divid[u]</td>
<td>Divide Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>7C000396</td>
<td>SR</td>
<td>B</td>
<td>divwu[u]</td>
<td>Divide Word</td>
</tr>
<tr>
<td>XFX</td>
<td>7C00039C</td>
<td>O</td>
<td>E.PM</td>
<td>mtpmr</td>
<td>Move To Performance Monitor Register</td>
</tr>
<tr>
<td>XFX</td>
<td>7C0003A6</td>
<td>O</td>
<td>B</td>
<td>mspsr</td>
<td>Move To Special Purpose Register</td>
</tr>
<tr>
<td>X</td>
<td>7C0003AC</td>
<td>P</td>
<td>E</td>
<td>dcbei</td>
<td>Data Cache Block Invalidate</td>
</tr>
<tr>
<td>X</td>
<td>7C0003CC</td>
<td>DS</td>
<td>dsn</td>
<td>Decorated Storage Notify</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003CC</td>
<td>M</td>
<td>ECL</td>
<td>icbitls</td>
<td>Instruction Cache Block Touch and Lock Set</td>
</tr>
<tr>
<td>X</td>
<td>7C0003CC</td>
<td>H</td>
<td>E.CD</td>
<td>dcread</td>
<td>Data Cache Read [Alternative Encoding]</td>
</tr>
<tr>
<td>XO</td>
<td>7C0003CE</td>
<td>V</td>
<td>stvx</td>
<td>Store Vector Indexed LRU</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C0003D2</td>
<td>SR</td>
<td>64</td>
<td>divid[u]</td>
<td>Divide Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>7C0003D6</td>
<td>SR</td>
<td>B</td>
<td>divwu[u]</td>
<td>Divide Word</td>
</tr>
<tr>
<td>X</td>
<td>7C000400</td>
<td>E</td>
<td>mcrxr</td>
<td>Move To Condition Register from XER</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000406</td>
<td>DS</td>
<td>lbdx</td>
<td>Load Byte with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00042A</td>
<td>MA</td>
<td>lswx</td>
<td>Load String Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00042C</td>
<td>B</td>
<td>lwbrx</td>
<td>Load Word Byte-Reverse Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000430</td>
<td>SR</td>
<td>B</td>
<td>srw[,]</td>
<td>Shift Right Word</td>
</tr>
<tr>
<td>X</td>
<td>7C000436</td>
<td>SR</td>
<td>64</td>
<td>srd[,]</td>
<td>Shift Right Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>7C000444</td>
<td>DS</td>
<td>lhdx</td>
<td>Load Halfword with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000446</td>
<td>H</td>
<td>E</td>
<td>tbsync</td>
<td>TLB Synchronize</td>
</tr>
<tr>
<td>X</td>
<td>7C000470</td>
<td>SR</td>
<td>VLE</td>
<td>e_srw[,]</td>
<td>Shift Right Word Immediate</td>
</tr>
<tr>
<td>X</td>
<td>7C000486</td>
<td>DS</td>
<td>lwdx</td>
<td>Load Word with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0004AA</td>
<td>MA</td>
<td>lswi</td>
<td>Load String Word Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0004AC</td>
<td>B</td>
<td>sync</td>
<td>Synchronize</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0004BE</td>
<td>P</td>
<td>E.PD</td>
<td>llfdepx</td>
<td>Load Floating-Point Double by External Process ID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C0004C6</td>
<td>DS</td>
<td>liddx</td>
<td>Load Doubleword with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000506</td>
<td>DS</td>
<td>stbdx</td>
<td>Store Byte with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00052A</td>
<td>MA</td>
<td>stswx</td>
<td>Store String Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00052C</td>
<td>B</td>
<td>stwbrx</td>
<td>Store Word Byte-Reverse Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000546</td>
<td>DS</td>
<td>sthdx</td>
<td>Store Halfword with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00056D</td>
<td>B</td>
<td>stbxc.</td>
<td>Store Byte Conditional Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000586</td>
<td>DS</td>
<td>stwdx</td>
<td>Store Word with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005AA</td>
<td>MA</td>
<td>stswi</td>
<td>Store String Word Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005AD</td>
<td>B</td>
<td>stbxc.</td>
<td>Store Halfword Conditional Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005BE</td>
<td>P</td>
<td>E.PD</td>
<td>stfdepx</td>
<td>Store Floating-Point Double by External Process ID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C0005C6</td>
<td>DS</td>
<td>stddev</td>
<td>Doubleword with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0005EC</td>
<td>E</td>
<td>dcbx</td>
<td>Data Cache Block Allocate</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Opcode (hexadecimal)²</td>
<td>Mode Dep. Priv.</td>
<td>Cat¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>-----------------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>X</td>
<td>7C00060E</td>
<td>P</td>
<td>E.PD</td>
<td>stvepxl</td>
<td>Store Vector by External Process ID Indexed LRU</td>
</tr>
<tr>
<td>X</td>
<td>7C000624</td>
<td>H</td>
<td>E</td>
<td>tlbivax</td>
<td>TLB Invalidate Virtual Address Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00062C</td>
<td>B</td>
<td>sraw[,]</td>
<td>lhbrx</td>
<td>Load Halfword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000634</td>
<td>SR</td>
<td>64</td>
<td>srad[,]</td>
<td>Shift Right Algebraic Word</td>
</tr>
<tr>
<td>EVX</td>
<td>7C00063E</td>
<td>P</td>
<td>E.PD</td>
<td>evldepx</td>
<td>Vector Load Doubleword into Doubleword by External Process ID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000646</td>
<td>DS</td>
<td>E</td>
<td>tlbdx</td>
<td>Load Floating Doubleword with Decoration Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00064E</td>
<td>P</td>
<td>E.PD</td>
<td>stvepx</td>
<td>Store Vector by External Process ID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000674</td>
<td>SR</td>
<td>64</td>
<td>srad[,]</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
</tr>
<tr>
<td>X</td>
<td>7C00067A</td>
<td>E</td>
<td>mbar</td>
<td>lhbrx</td>
<td>Load Halfword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00067C</td>
<td>H</td>
<td>E</td>
<td>tlbx</td>
<td>TLB Search Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000734</td>
<td>SR</td>
<td>64</td>
<td>srad[,]</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>7C00073E</td>
<td>P</td>
<td>E.PD</td>
<td>evstdepx</td>
<td>Vector Store Doubleword into Doubleword by External Process ID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C000746</td>
<td>DS</td>
<td>E</td>
<td>slfddx</td>
<td>Store Floating Doubleword with Decoration Indexed</td>
</tr>
<tr>
<td>X</td>
<td>7C00076A</td>
<td>H</td>
<td>E</td>
<td>tlbre</td>
<td>TLB Read Entry</td>
</tr>
<tr>
<td>X</td>
<td>7C000774</td>
<td>SR</td>
<td>B</td>
<td>extbs[,]</td>
<td>Extend Sign Byte</td>
</tr>
<tr>
<td>X</td>
<td>7C00078C</td>
<td>E,C</td>
<td>ici</td>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
</tr>
<tr>
<td>X</td>
<td>7C00079A</td>
<td>H</td>
<td>E</td>
<td>tlbw</td>
<td>TLB Write Entry</td>
</tr>
<tr>
<td>X</td>
<td>7C00079B</td>
<td>SR</td>
<td>64</td>
<td>extsw[,]</td>
<td>Extend Sign Word</td>
</tr>
<tr>
<td>X</td>
<td>7C0007BB</td>
<td>P</td>
<td>E.PD</td>
<td>icbiep</td>
<td>Instruction Cache Block Invalidate by External PID</td>
</tr>
<tr>
<td>X</td>
<td>7C0007CC</td>
<td>H</td>
<td>E,CD</td>
<td>icread</td>
<td>Instruction Cache Read</td>
</tr>
<tr>
<td>X</td>
<td>7C0007EC</td>
<td>B</td>
<td>E</td>
<td>dbcz</td>
<td>Data Cache Block set to Zero</td>
</tr>
<tr>
<td>X</td>
<td>7C0007CE</td>
<td>P</td>
<td>E.PD</td>
<td>dcbzep</td>
<td>Data Cache Block set to Zero by External PID</td>
</tr>
<tr>
<td>XFX</td>
<td>7C100026</td>
<td>B</td>
<td>mtocrf</td>
<td>Move From One Condition Register Field</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C100120</td>
<td>B</td>
<td>mtocrf</td>
<td>Move To One Condition Register Field</td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>8000-----</td>
<td>VLE</td>
<td>se_lbz</td>
<td>Load Byte and Zero Short Form</td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>9000-----</td>
<td>VLE</td>
<td>se_stb</td>
<td>Store Byte Short Form</td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>A000-----</td>
<td>VLE</td>
<td>se_lhz</td>
<td>Load Halfword and Zero Short Form</td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>B000-----</td>
<td>VLE</td>
<td>se_sth</td>
<td>Store Halfword Short Form</td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>C000-----</td>
<td>VLE</td>
<td>se_lwz</td>
<td>Load Word and Zero Short Form</td>
<td></td>
</tr>
<tr>
<td>SD4</td>
<td>D000-----</td>
<td>VLE</td>
<td>se_stw</td>
<td>Store Word Short Form</td>
<td></td>
</tr>
<tr>
<td>BD8</td>
<td>E000-----</td>
<td>VLE</td>
<td>se_bc</td>
<td>Branch Conditional Short Form</td>
<td></td>
</tr>
<tr>
<td>BD8</td>
<td>E800-----</td>
<td>VLE</td>
<td>se_bl[l]</td>
<td>Branch [and Link]</td>
<td></td>
</tr>
</tbody>
</table>

¹ See the key to the mode dependency and privilege column below and the key to the category column in Section 1.3.5 of Book I.

²For 16-bit instructions, the “Opcode” column represents the 16-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits; dashes are used following the opcode to indicate the form is a 16-bit instruction. For 32-bit instructions, the “Opcode” column represents the 32-bit hexadecimal instruction encoding with the opcode, extended opcode, and other fields with fixed values in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode, extended opcode or fixed value bits.

Mode Dependency and Privilege Abbreviations

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

Mode Dep. Description

CT If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.

SR The setting of status registers (such as XER and CR0) is mode-dependent.
Mode Dep. Description

32 The instruction must be executed only in 32-bit mode.

64 The instruction must be executed only in 64-bit mode.

Key to Privilege Column

Priv. Description

P Denotes a privileged instruction.

O Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspri), depending on the SPR or PMR number.

M Denotes an instruction that is treated as privileged or nonprivileged, depending on the value of the UCLE bit of the MSR.

H Denotes an instruction that can be executed only in hypervisor state.
Appendices:

Power ISA Book I-III Appendices
Appendix A. Incompatibilities with the POWER Architecture

This appendix identifies the known incompatibilities that must be managed in the migration from the POWER Architecture to the Power ISA. Some of the incompatibilities can, at least in principle, be detected by the processor, which could trap and let software simulate the POWER operation. Others cannot be detected by the processor even in principle.

In general, the incompatibilities identified here are those that affect a POWER application program. Incompatibilities for instructions that can be used only by POWER operating system programs are not necessarily discussed. Discussion of incompatibilities that pertain only to operating system programs assumes the Server environment (because there is no need for POWER operating system programs to run in the Embedded environment).

A.1 New Instructions, Formerly Privileged Instructions

Instructions new to Power ISA typically use opcode values (including extended opcode) that are illegal in POWER. A few instructions that are privileged in POWER (e.g., 
\[ \text{dclz} \] called \[ \text{dcbz} \] in Power ISA) have been made nonprivileged in Power ISA. Any POWER program that executes one of these now-valid or now-nonprivileged instructions, expecting to cause the system illegal instruction error handler or the system privileged instruction error handler to be invoked, will not execute correctly on Power ISA.

A.2 Newly Privileged Instructions

The following instructions are nonprivileged in POWER but privileged in Power ISA.
- `mfmsr`
- `mfsr`

A.3 Reserved Fields in Instructions

These fields are shown with "/"s in the instruction layouts. In both POWER and Power ISA these fields are ignored by the processor. The Power ISA states that these fields must contain zero. The POWER Architecture lacks such a statement, but it is expected that essentially all POWER programs contain zero in these fields.

In several cases the Power ISA assumes that reserved fields in POWER instructions indeed contain zero. The cases include the following.
- `bclr[l]` and `bcctr[l]` assume that bits 19:20 in the POWER instructions contain zero.
- `cmpi`, `cmp`, `cmpli`, and `cmpl` assume that bit 10 in the POWER instructions contains zero.
- `mtspr` and `mfspr` assume that bits 16:20 in the POWER instructions contain zero.
- `mtcrf` and `mfcr` assume that bit 11 in the POWER instructions contains zero.
- Synchronize assumes that bits 9:10 in the POWER instruction (\[ \text{dcs} \]) contain zero. (This assumption provides compatibility for application programs, but not necessarily for operating system programs; see Section A.22.)
- `mtmsr` assumes that bit 15 in the POWER instruction contains zero.

A.4 Reserved Bits in Registers

Both POWER and Power ISA permit software to write any value to these bits. However in POWER reading such a bit always returns 0, while in Power ISA reading it may return either 0 or the value that was last written to it.

A.5 Alignment Check

The POWER MSR \[ \text{AL} \] bit (bit 24) is no longer supported; the corresponding Power ISA MSR bit, bit 56, is reserved. The low-order bits of the EA are always used. (Notice that the value 0 — the normal value for a reserved bit — means "ignore the low-order EA bits" in
POWER, and the value 1 means "use the low-order EA bits"). POWER-compatible operating system code will probably write the value 1 to this bit.

A.6 Condition Register

The following instructions specify a field in the CR explicitly (via the BF field) and also, in POWER, use bit 31 as the Record bit. In Power ISA, bit 31 is a reserved field for these instructions and is ignored by the processor. In POWER, if bit 31 contains 1 the instructions execute normally (i.e., as if the bit contained 0) except as follows:

- `cmp` CR0 is undefined if Rc=1 and BF≠0
- `cmpl` CR0 is undefined if Rc=1 and BF≠0
- `mcrxr` CR0 is undefined if Rc=1 and BF≠0
- `fcmpu` CR1 is undefined if Rc=1
- `fcmpo` CR1 is undefined if Rc=1
- `mcrfs` CR1 is undefined if Rc=1 and BF≠1

A.7 LK and Rc Bits

For the instructions listed below, if bit 31 (LK or Rc bit in POWER) contains 1, in POWER the instruction executes as if the bit contained 0 except as follows: if LK=1, the Link Register is set (to an undefined value, except for `svc`); if Rc=1, Condition Register Field 0 or 1 is set to an undefined value. In Power ISA, bit 31 is a reserved field for these instructions and is ignored by the processor.

Power ISA instructions for which bit 31 is the LK bit in POWER:

- `sc` (svc in POWER)
- the Condition Register Logical instructions
- `mcrf`
- `isync` (ics in POWER)

Power ISA instructions for which bit 31 is the Rc bit in POWER:

- fixed-point X-form Load and Store instructions
- fixed-point X-form Compare instructions
- the X-form Trap instruction
- `mtspr`, `mfspr`, `mtcrf`, `mcrf`, `mtocrf`, `mtocrf`
- floating-point X-form Load and Store instructions
- floating-point Compare instructions
- `mcrfs`
- `dcbz` (dclz in POWER)

A.8 BO Field

POWER shows certain bits in the BO field — used by Branch Conditional instructions — as "x". Although the POWER Architecture does not say how these bits are to be interpreted, they are in fact ignored by the processor.

Power ISA shows these bits as "z", "a", or "t". The "z" bits are ignored, as in POWER. However, the "a" and "t" bits can be used by software to provide a hint about how the branch is likely to behave. If a POWER program has the "wrong" value for these bits, the program will produce the same results as on POWER but performance may be affected.

A.9 BH Field

Bits 19:20 of the Branch Conditional to Link Register and Branch Conditional to Count Register instructions are reserved in POWER but are defined as a branch hint (BH) field in Power ISA. Because these bits are hints, they may affect performance but do not affect the results of executing the instruction.

A.10 Branch Conditional to Count Register

For the case in which the Count Register is decremented and tested (i.e., the case in which BO₂=0), POWER specifies only that the branch target address is undefined, with the implication that the Count Register, and the Link Register if LK=1, are updated in the normal way. Power ISA specifies that this instruction form is invalid.

A.11 System Call

There are several respects in which Power ISA is incompatible with POWER for System Call instructions — which in POWER are called Supervisor Call instructions.

- POWER provides a version of the Supervisor Call instruction (bit 30 = 0) that allows instruction fetching to continue at any one of 128 locations. It is used for "fast SVCs". Power ISA provides no such version: if bit 30 of the instruction is 0 the instruction form is invalid.
- POWER provides a version of the Supervisor Call instruction (bits 30:31 = 0b11) that resumes instruction fetching at one location and sets the Link Register to the address of the next instruction. Power ISA provides no such version: bit 31 is a reserved field.
- For POWER, information from the MSR is saved in the Count Register. For Power ISA this information is saved in SRR1.
- In POWER bits 16:19 and 27:29 of the instruction comprise defined instruction fields or a portion thereof, while in Power ISA these bits comprise reserved fields.
In POWER bits 20:26 of the instruction comprise a portion of the SV field, while in Power ISA these bits comprise the LEV field.

POWER saves the low-order 16 bits of the instruction, in the Count Register. Power ISA does not save them.

The settings of MSR bits by the associated interrupt differ between POWER and Power ISA; see POWER Processor Architecture and Book III.

A.12 Fixed-Point Exception Register (XER)

Bits 48:55 of the XER are reserved in Power ISA, while in POWER the corresponding bits (16:23) are defined and contain the comparison byte for the \texttt{lscbx} instruction (which Power ISA lacks).

A.13 Update Forms of Storage Access Instructions

Power ISA requires that RA not be equal to either RT (fixed-point Load only) or 0. If the restriction is violated the instruction form is invalid. POWER permits these cases, and simply avoids saving the EA.

A.14 Multiple Register Loads

Power ISA requires that RA, and RB if present in the instruction format, not be in the range of registers to be loaded, while POWER permits this and does not alter RA or RB in this case. (The Power ISA restriction applies even if RA=0, although there is no obvious benefit to the restriction in this case since RA is not used to compute the effective address if RA=0.) If the Power ISA restriction is violated, either the system illegal instruction error handler is invoked or the results are boundedly undefined. The instructions affected are:

- \texttt{lmw} (\texttt{lm} in POWER)
- \texttt{lswi} (\texttt{lsi} in POWER)
- \texttt{lswx} (\texttt{lsx} in POWER)

For example, an \texttt{lmw} instruction that loads all 32 registers is valid in POWER but is an invalid form in Power ISA.

A.15 Load/Store Multiple Instructions

There are two respects in which Power ISA is incompatible with POWER for Load Multiple and Store Multiple instructions.

If the EA is not word-aligned, in Power ISA either an Alignment exception occurs or the addressed bytes are loaded, while in POWER an Alignment interrupt occurs if \texttt{MSRAL=1} (the low-order two bits of the EA are ignored if \texttt{MSRAL=0}).

In Power ISA the instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

A.16 Move Assist Instructions

There are several respects in which Power ISA is incompatible with POWER for Move Assist instructions.

- In Power ISA an \texttt{lswx} instruction with zero length leaves the contents of RT undefined (if RT≠RA and RT≠RB) or is an invalid instruction form (if RT=RA or RT=RB), while in POWER the corresponding instruction (\texttt{lsx}) is a no-op in these cases.

In Power ISA a Move Assist instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

A.17 Move To/From SPR

There are several respects in which Power ISA is incompatible with POWER for Move To/From Special Purpose Register instructions.

- The SPR field is ten bits long in Power ISA, but only five in POWER (see also Section A.3, “Reserved Fields in Instructions”).
- \texttt{mfspr} can be used to read the Decrementer in problem state in POWER, but only in privileged state in Power ISA.

If the SPR value specified in the instruction is not one of the defined values, POWER behaves as follows.
- If the instruction is executed in problem state and \texttt{SPR0=1}, a Privileged Instruction type Program interrupt occurs. No architected registers are altered except those set by the interrupt.
- Otherwise no architected registers are altered.

In this same case, Power ISA behaves as follows.
- If the instruction is executed in problem state, a Hypervisor Emulation Assistance interrupt occurs if \texttt{spr0=0} and a Privileged Instruction type Program interrupt occurs if \texttt{spr0=1}. No architected registers are altered except those set by the interrupt.
- If the instruction is executed in privileged state, a Hypervisor Emulation Assistance interrupt occurs if the SPR value is 0 or, for \texttt{mfspr} only, if the SPR value is 4, 5, or 6. In
these cases no architected registers are altered except those set by the interrupt. Otherwise no operation is performed. (See Section 4.4.4, “Move To/From System Register Instructions” in Book III-S.)

A.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, POWER does not specify how FR and FI are set, while Power ISA preserves them for Invalid Operation Exception caused by a Compare instruction, sets FI to 1 and FR to an undefined value for disabled Overflow Exception, and clears them otherwise.

- Invalid Operation Exception (enabled or disabled)
- Zero Divide Exception (enabled or disabled)
- Disabled Overflow Exception

A.19 Store Floating-Point Single Instructions

There are several respects in which Power ISA is incompatible with POWER for Store Floating-Point Single instructions.

- POWER uses FPSCR\textsubscript{UE} to help determine whether denormalization should be done, while Power ISA does not. Using FPSCR\textsubscript{UE} is in fact incorrect: if FPSCR\textsubscript{UE}=1 and a denormalized single-precision number is copied from one storage location to another by means of \textit{lfs} followed by \textit{stfs}, the two “copies” may not be the same.

- For an operand having an exponent that is less than 874 (unbiased exponent less than \textasciitilde -149), POWER stores a zero (if FPSCR\textsubscript{UE}=0) while Power ISA stores an undefined value.

A.20 Move From FPSCR

POWER defines the high-order 32 bits of the result of \textit{mfs} to be 0xFFFF_FFFF, while Power ISA copies the high-order 32-bits of the FPSCR.

A.21 Zeroing Bytes in the Data Cache

The \textit{dclz} instruction of POWER and the \textit{dcbz} instruction of Power ISA have the same opcode. However, the functions differ in the following respects.

- \textit{dclz} clears a line while \textit{dcbz} clears a block.
- \textit{dclz} saves the EA in RA (if RA\neq 0) while \textit{dcbz} does not.
- \textit{dclz} is privileged while \textit{dcbz} is not.

A.22 Synchronization

The \textit{Synchronize} instruction (called \textit{dcs} in POWER) and the \textit{isync} instruction (called \textit{ics} in POWER) cause more pervasive synchronization in Power ISA than in POWER. However, unlike \textit{dcs}, \textit{Synchronize} does not wait until data cache block writes caused by preceding instructions have been performed in main storage. Also, \textit{Synchronize} has an L field while \textit{dcs} does not, and some uses of the instruction by the operating system require L=2<S>. (The L field corresponds to reserved bits in \textit{dcs} and hence is expected to be zero in POWER programs; see Section A.3.)

A.23 Move To Machine State Register Instruction

The \textit{mtmsr} instruction has an L field in Power ISA but not in POWER. The function of the variant of \textit{mtmsr} with L=1 differs from the function of the instruction in the POWER architecture in the following ways.

- In Power ISA, this variant of \textit{mtmsr} modifies only the EE and RI bits of the MSR, while in the POWER \textit{mtmsr} modifies all bits of the MSR.

- This variant of \textit{mtmsr} is execution synchronizing in Power ISA but is context synchronizing in POWER. (The POWER architecture lacks Power ISA’s distinction between execution synchronization and context synchronization. The statement in the POWER architecture specification that \textit{mtmsr} is “synchronizing” is equivalent to stating that the instruction is context synchronizing.)

Also, \textit{mtmsr} is optional in Power ISA but required in POWER.

A.24 Direct-Store Segments

POWER’s direct-store segments are not supported in Power ISA.

A.25 Segment Register Manipulation Instructions

The definitions of the four Segment Register Manipulation instructions \textit{mtsr}, \textit{mtsri}, \textit{mfsr}, and \textit{mfsri} differ in two respects between POWER and Power ISA. Instructions similar to \textit{mtsri} and \textit{mfsri} are called \textit{mtsri} and \textit{mfsri} in POWER.

- \textit{mfsr} and \textit{mfsri} are problem state instructions in POWER, while \textit{mfsr} and \textit{mfsri} are privileged in Power ISA.

- function: the “indirect” instructions (\textit{mtsri} and \textit{mfsri}) in POWER use an RA register in computing the Segment Register number, and the computed EA is stored into RA (if
Appendix A. Incompatibilities with the POWER Architecture

A.26 TLB Entry Invalidation

The `tlbi` instruction of POWER and the `tlbie` instruction of Power ISA have the same opcode. However, the functions differ in the following respects.

- `tlbi` computes the EA as \((RA) \land 0 + (RB)\), while `tlbie` lacks an RA field and computes the EA and related information as \((RB)\).
- `tlbi` saves the EA in RA (if RA\(\neq 0\)), while `tlbie` lacks an RA field and does not save the EA.
- For `tlbi` the high-order 36 bits of RB are used in computing the EA, while for `tlbie` these bits contain additional information that is not directly related to the EA.
- For `tlbi` has no RS operand, while for `tlbie` the \(RS\) is an LPID value used to qualify the TLB invalidation.

Also, `tlbi` is required in POWER whereas `tlbie` is optional in Power ISA.

A.27 Alignment Interrupts

Placing information about the interrupting instruction into the DSISR and the DAR when an Alignment interrupt occurs is optional in Power ISA but required in POWER.

A.28 Floating-Point Interrupts

POWER uses MSR bit 20 to control the generation of interrupts for floating-point enabled exceptions, and Power ISA uses the corresponding MSR bit, bit 52, for the same purpose. However, in Power ISA this bit is part of a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER this bit is used independently to control the occurrence of the interrupt (in POWER all floating-point interrupts are precise).

A.29 Timing Facilities

A.29.1 Real-Time Clock

The POWER Real-Time Clock is not supported in Power ISA. Instead, Power ISA provides a Time Base. Both the RTC and the TB are 64-bit Special Purpose Registers, but they differ in the following respects.

- The RTC counts seconds and nanoseconds, while the TB counts “ticks”. The ticking rate of the TB is implementation-dependent.
- The RTC increments discontinuously: 1 is added to RTCU when the value in RTCL passes 999,999,999. The TB increments continuously: 1 is added to TBU when the value in TBL passes 0xFFFF_FFFF.
- The RTC is written and read by the `mtspr` and `mfspr` instructions, using SPR numbers that denote the RTCU and RTCL. The TB is written and read by the same instructions using different SPR numbers.
- The SPR numbers that denote POWER's RTCL and RTCU are invalid in Power ISA.
- The RTC is guaranteed to increment at least once in the time required to execute ten \textit{Add Immediate} instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

A.29.2 Decrementer

The Power ISA Decrementer differs from the POWER Decrementer in the following respects.

- The Power ISA DEC decrements at the same rate that the TB increments, while the POWER DEC decrements every nanosecond (which is the same rate that the RTC increments).
- Not all bits of the POWER DEC need be implemented, while all bits of the Power ISA DEC must be implemented.
- The interrupt caused by the DEC has its own interrupt vector location in Power ISA, but is considered an External interrupt in POWER.
A.30 Deleted Instructions

The following instructions are part of the POWER Architecture but have been dropped from the Power ISA.

<table>
<thead>
<tr>
<th>MNEM</th>
<th>PRI</th>
<th>XOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>31</td>
<td>360</td>
</tr>
<tr>
<td>clcs</td>
<td>31</td>
<td>531</td>
</tr>
<tr>
<td>clf</td>
<td>31</td>
<td>118</td>
</tr>
<tr>
<td>cli (*)</td>
<td>31</td>
<td>502</td>
</tr>
<tr>
<td>dcist</td>
<td>31</td>
<td>630</td>
</tr>
<tr>
<td>div</td>
<td>31</td>
<td>331</td>
</tr>
<tr>
<td>divs</td>
<td>31</td>
<td>363</td>
</tr>
<tr>
<td>doz</td>
<td>31</td>
<td>264</td>
</tr>
<tr>
<td>dozi</td>
<td>09</td>
<td>-</td>
</tr>
<tr>
<td>lscbx</td>
<td>31</td>
<td>277</td>
</tr>
<tr>
<td>maskg</td>
<td>31</td>
<td>29</td>
</tr>
<tr>
<td>maskir</td>
<td>31</td>
<td>541</td>
</tr>
<tr>
<td>mfsri</td>
<td>31</td>
<td>627</td>
</tr>
<tr>
<td>mul</td>
<td>31</td>
<td>107</td>
</tr>
<tr>
<td>nabs</td>
<td>31</td>
<td>488</td>
</tr>
<tr>
<td>rac (*)</td>
<td>31</td>
<td>818</td>
</tr>
<tr>
<td>rfi (*)</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>rfsvc</td>
<td>19</td>
<td>82</td>
</tr>
<tr>
<td>rli</td>
<td>22</td>
<td>-</td>
</tr>
<tr>
<td>rrib</td>
<td>31</td>
<td>537</td>
</tr>
<tr>
<td>sle</td>
<td>31</td>
<td>153</td>
</tr>
<tr>
<td>sleq</td>
<td>31</td>
<td>217</td>
</tr>
<tr>
<td>sliq</td>
<td>31</td>
<td>184</td>
</tr>
<tr>
<td>sliq</td>
<td>31</td>
<td>248</td>
</tr>
<tr>
<td>sliq</td>
<td>31</td>
<td>216</td>
</tr>
<tr>
<td>slq</td>
<td>31</td>
<td>152</td>
</tr>
<tr>
<td>sraiq</td>
<td>31</td>
<td>952</td>
</tr>
<tr>
<td>sraq</td>
<td>31</td>
<td>920</td>
</tr>
<tr>
<td>sre</td>
<td>31</td>
<td>665</td>
</tr>
<tr>
<td>srea</td>
<td>31</td>
<td>921</td>
</tr>
<tr>
<td>sreq</td>
<td>31</td>
<td>729</td>
</tr>
<tr>
<td>sriq</td>
<td>31</td>
<td>696</td>
</tr>
<tr>
<td>sriq</td>
<td>31</td>
<td>760</td>
</tr>
<tr>
<td>sriq</td>
<td>31</td>
<td>728</td>
</tr>
<tr>
<td>sq</td>
<td>31</td>
<td>664</td>
</tr>
</tbody>
</table>

(*) This instruction is privileged.

Note: Many of these instructions use the MQ register. The MQ is not defined in the Power ISA.

A.31 Discontinued Opcodes

The opcodes listed below are defined in the POWER Architecture but have been dropped from the Power ISA. The list contains the POWER mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The corresponding instructions are reserved in Power ISA.

Assembler Note

It might be helpful to current software writers for the Assembler to flag the discontinued POWER instructions.
A.32 POWER2 Compatibility

The POWER2 instruction set is a superset of the POWER instruction set. Some of the instructions added for POWER2 are included in the Power ISA. Those that have been renamed in the Power ISA are listed in this section, as are the new POWER2 instructions that are not included in the Power ISA. Other incompatibilities are also listed.

A.32.1 Cross-Reference for Changed POWER2 Mnemonics

The following table lists the new POWER2 instruction mnemonics that have changed in the Power ISA User Instruction Set Architecture, sorted by POWER2 mnemonic.

To determine the Power ISA mnemonic for one of these POWER2 mnemonics, find the POWER2 mnemonic in the second column of the table: the remainder of the line gives the Power ISA mnemonic and the page on which the instruction is described, as well as the instruction names.

POWER2 mnemonics that have not changed are not listed.

<table>
<thead>
<tr>
<th>Page</th>
<th>POWER2 Mnemonic</th>
<th>POWER2 Instruction</th>
<th>Power ISA Mnemonic</th>
<th>Power ISA Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>fcir[.]</td>
<td>Floating Convert Double to Integer with Round</td>
<td>fctiw[.]</td>
<td>Floating Convert To Integer Word</td>
</tr>
<tr>
<td>154</td>
<td>fcirz[.]</td>
<td>Floating Convert Double to Integer with Round to Zero</td>
<td>fctiwz[.]</td>
<td>Floating Convert To Integer Word with round toward Zero</td>
</tr>
</tbody>
</table>

A.32.2 Load/Store Floating-Point Double

Several of the opcodes for the Load/Store Floating-Point Quad instructions of the POWER2 architecture have been reclaimed by the Load/Store Floating-Point Double [Indexed] instructions (entries with a ‘-’ in the Power ISA column have not been reclaimed):

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>POWER2</th>
<th>POWER ISA</th>
<th>PRI</th>
<th>XOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfpq</td>
<td>lq</td>
<td>56</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>lfpqu</td>
<td>lfdp</td>
<td>57</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lfpqx</td>
<td>-</td>
<td>31</td>
<td>823</td>
<td></td>
</tr>
<tr>
<td>stfq</td>
<td>-</td>
<td>60</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>stfqu</td>
<td>stfdp</td>
<td>61</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>stfqux</td>
<td>-</td>
<td>31</td>
<td>951</td>
<td></td>
</tr>
<tr>
<td>stfqx</td>
<td>stfdpx</td>
<td>31</td>
<td>919</td>
<td></td>
</tr>
</tbody>
</table>

Differences between the lfstfdp[x] instructions and the POWER2 lfstfq[u][x] instructions include the following:

- The storage operand for the lfstfdp[x] instructions must be quadword aligned for optimal performance.
- The register pairs for the lfstfdp[x] instructions must be even-odd pairs, instead of any consecutive pair.
- The lfstfdp[x] instructions do not have update forms.

A.32.3 Floating-Point Conversion to Integer

The fcir and fcirz instructions of POWER2 have the same opcodes as do the fctiw and fctiwz instructions, respectively, of Power ISA. However, the functions differ in the following respects.

- fcir and fcirz set the high-order 32 bits of the target FPR to 0xFFFF_FFFF, while fctiw and fctiwz set them to an undefined value.
- Except for enabled Invalid Operation Exceptions, fcir and fcirz set the FPREF field of the FPSCR based on the result, while fctiw and fctiwz set it to an undefined value.
- fcir and fcirz do not affect the VXSNAN bit of the FPSCR, while fctiw and fctiwz do.
- fcir and fcirz set FPSCR.X to 1 for certain cases of “Large Operands” (i.e., operands that are too large to be represented as a 32-bit signed fixed-point integer), while fctiw and fctiwz do not alter it for any case of “Large Operands”. (The IEEE standard requires not altering it for “Large Operands”.)
A.32.4 Floating-Point Interrupts

POWER2 uses MSR bits 20 and 23 to control the generation of interrupts for floating-point enabled exceptions, and Power ISA uses the corresponding MSR bits, bits 52 and 55, for the same purpose. However, in Power ISA these bits comprise a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER2 these bits are used independently to control the occurrence (bit 20) and the precision (bit 23) of the interrupt. Moreover, in Power ISA all floating-point interrupts are considered Program interrupts, while in POWER2 imprecise floating-point interrupts have their own interrupt vector location.

A.32.5 Trace

The Trace interrupt vector location differs between the two architectures, and there are many other differences.

A.33 Deleted Instructions

The following instructions are new in POWER2 implementations of the POWER Architecture but have been dropped from the Power ISA.

<table>
<thead>
<tr>
<th>MNEM</th>
<th>Pri</th>
<th>XOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfq</td>
<td>56</td>
<td>-</td>
</tr>
<tr>
<td>lfqu</td>
<td>31</td>
<td>791</td>
</tr>
<tr>
<td>lfqux</td>
<td>31</td>
<td>919</td>
</tr>
<tr>
<td>stfq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfqu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stfqx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A.33.1 Discontinued Opcodes

The opcodes listed below are new in POWER2 implementations of the POWER Architecture but have been dropped from the Power ISA. The list contains the POWER2 mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The instructions are either illegal or reserved in Power ISA; see Appendix D.
Appendix B. Platform Support Requirements

As described in Chapter 1 of Book I, the architecture is structured as a collection of categories. Each category is comprised of facilities and/or instructions that together provide a unit of functionality. The Server and Embedded categories are referred to as "special" because all implementations must support at least one of these categories. Each special category, when taken together with the Base category, is referred to as an "environment", and provides the minimum functionality required to develop operating systems and applications.

Every processor implementation supports at least one of the environments, and may also support a set of categories chosen based on the target market for the implementation. However, a Server implementation supports only those categories designated as part of the Server platform in Figure 1. To facilitate the development of operating systems and applications for a well-defined purpose or customer set, usually embodied in a unique hardware platform, this appendix documents the association between a platform and the set of categories it requires.

Adding a new platform may permit cost-performance optimization by clearly identifying a unique set of categories. However, this has the potential to fragment the application base. As a result, new platforms will be added only when the optimization benefit clearly outweighs the loss due to fragmentation.

The platform support requirements are documented in Figure 1. An “x” in a column indicates that the category is required. A “+” in a column indicates that the requirement is being phased in.
<table>
<thead>
<tr>
<th>Category</th>
<th>Server Platform</th>
<th>Embedded Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Server</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Embedded</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Alternate Time Base</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache Specification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal Floating-Point</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Decorated Storage</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Elemental Memory Barriers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Cache Debug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Cache Initialization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Device Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Enhanced Debug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Hypervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Hypervisor.LRAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Little-Endian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Page Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Performance Monitor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Processor Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.Cache Locking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded Multi-Threading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded Multi-Threading.Thread Manage-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embedded.TLB Write Conditional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Proxy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-Point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-Point.Record</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Legacy Move Assist</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Legacy Integer Multiply-Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store Quadword</td>
<td>x²</td>
<td></td>
</tr>
<tr>
<td>Memory Coherence</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Move Assist</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Processor Compatibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal Processing Engine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPE.Embedded Float Scalar Double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPE.Embedded Float Scalar Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPE.Embedded Float Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Conditional Page Mobility</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Strong Access Order</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Trace</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Transactional Memory</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Figure 1. Platform Support Requirements (Sheet 1 of 2)
<table>
<thead>
<tr>
<th>Category</th>
<th>Server Platform</th>
<th>Embedded Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Length Encoding</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Vector.Little-Endian</td>
<td>+, 1</td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-Bit</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

1. If the Vector category is supported, Vector.Little-Endian is required on the Server platform.
2. Optional for the Server Platform.

Figure 1. Platform Support Requirements (Sheet 2 of 2)
# Appendix C. Complete SPR List

This appendix lists all the Special Purpose Registers in the Power ISA, ordered by SPR number.

<table>
<thead>
<tr>
<th>decimal</th>
<th>spr5:9</th>
<th>spr0:4</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000</td>
<td>00001</td>
<td>XER</td>
<td>no</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>00000</td>
<td>00011</td>
<td>DSCHR</td>
<td>no</td>
<td>64</td>
<td>STM</td>
</tr>
<tr>
<td>8</td>
<td>00000</td>
<td>01000</td>
<td>LR</td>
<td>no</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>00000</td>
<td>01001</td>
<td>CTR</td>
<td>no</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>13</td>
<td>00000</td>
<td>01101</td>
<td>AMR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>17</td>
<td>00000</td>
<td>10001</td>
<td>DSCR</td>
<td>yes</td>
<td>64</td>
<td>STM</td>
</tr>
<tr>
<td>18</td>
<td>00000</td>
<td>10010</td>
<td>DSISR</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>19</td>
<td>00000</td>
<td>10011</td>
<td>DAR</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>22</td>
<td>00000</td>
<td>10110</td>
<td>DEC</td>
<td>yes13</td>
<td>32</td>
<td>B</td>
</tr>
<tr>
<td>25</td>
<td>00000</td>
<td>11001</td>
<td>SDR1</td>
<td>hypv13</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>26</td>
<td>00000</td>
<td>11010</td>
<td>SRR0</td>
<td>hypv13</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>27</td>
<td>00000</td>
<td>11011</td>
<td>SRR1</td>
<td>hypv13</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>28</td>
<td>00000</td>
<td>11100</td>
<td>CFAR</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>29</td>
<td>00000</td>
<td>11101</td>
<td>AMR</td>
<td>yes13</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>48</td>
<td>00001</td>
<td>10000</td>
<td>PID</td>
<td>yes</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>53</td>
<td>00001</td>
<td>10101</td>
<td>GDECAR</td>
<td>hypv13</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>54</td>
<td>00001</td>
<td>10110</td>
<td>DECAR</td>
<td>hypv13</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>55</td>
<td>00001</td>
<td>10111</td>
<td>MCIVPR</td>
<td>hypv12</td>
<td>hypv12</td>
<td>64</td>
</tr>
<tr>
<td>56</td>
<td>00001</td>
<td>11000</td>
<td>LPER</td>
<td>hypv12</td>
<td>hypv12</td>
<td>64</td>
</tr>
<tr>
<td>57</td>
<td>00001</td>
<td>11001</td>
<td>LPERU</td>
<td>hypv12</td>
<td>hypv12</td>
<td>32</td>
</tr>
<tr>
<td>58</td>
<td>00001</td>
<td>11100</td>
<td>CSRR0</td>
<td>hypv12</td>
<td>hypv12</td>
<td>64</td>
</tr>
<tr>
<td>59</td>
<td>00001</td>
<td>11101</td>
<td>CSRR1</td>
<td>hypv12</td>
<td>hypv12</td>
<td>32</td>
</tr>
<tr>
<td>60</td>
<td>00001</td>
<td>11110</td>
<td>GTSRWR</td>
<td>hypv13</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>61</td>
<td>00001</td>
<td>11111</td>
<td>IAMR</td>
<td>yes13</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>62</td>
<td>00001</td>
<td>11111</td>
<td>ESR</td>
<td>yes13</td>
<td>yes13</td>
<td>32</td>
</tr>
<tr>
<td>63</td>
<td>00001</td>
<td>11111</td>
<td>IVPR</td>
<td>hypv13</td>
<td>hypv12</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>00100</td>
<td>00000</td>
<td>TFHAR</td>
<td>no</td>
<td>64</td>
<td>TM</td>
</tr>
<tr>
<td>129</td>
<td>00100</td>
<td>00001</td>
<td>TFIAR</td>
<td>no</td>
<td>64</td>
<td>TM</td>
</tr>
<tr>
<td>130</td>
<td>00100</td>
<td>00010</td>
<td>TEXASR</td>
<td>no</td>
<td>64</td>
<td>TM</td>
</tr>
<tr>
<td>131</td>
<td>00100</td>
<td>00011</td>
<td>TEXASRU</td>
<td>no</td>
<td>32</td>
<td>TM</td>
</tr>
<tr>
<td>136</td>
<td>00100</td>
<td>01000</td>
<td>CTRL</td>
<td>-</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>152</td>
<td>00100</td>
<td>11000</td>
<td>CTRL</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>153</td>
<td>00100</td>
<td>11001</td>
<td>FSCR</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>157</td>
<td>00100</td>
<td>11100</td>
<td>UAMOR</td>
<td>yes13</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>159</td>
<td>00100</td>
<td>11111</td>
<td>PSPB</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>176</td>
<td>00101</td>
<td>00000</td>
<td>DPDES</td>
<td>hypv13</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>177</td>
<td>00101</td>
<td>01000</td>
<td>DHDES</td>
<td>hypv13</td>
<td>hypv13</td>
<td>64</td>
</tr>
<tr>
<td>180</td>
<td>00101</td>
<td>10100</td>
<td>DAWR0</td>
<td>hypv13</td>
<td>hypv13</td>
<td>64</td>
</tr>
</tbody>
</table>

Figure 2. SPR Numbers (Sheet 1 of 5)
<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR^1</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>186</td>
<td>00101 11010</td>
<td>RPR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>187</td>
<td>00101 11011</td>
<td>CIABR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>188</td>
<td>00101 11100</td>
<td>DAWRX0</td>
<td>hypv^3</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>190</td>
<td>00101 11101</td>
<td>HFSCR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>256</td>
<td>01000 00000</td>
<td>VRSAVE</td>
<td>no</td>
<td>32</td>
<td>B</td>
</tr>
<tr>
<td>259</td>
<td>01000 00011</td>
<td>SPRG3</td>
<td>-</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>260-263</td>
<td>01000 001xx</td>
<td>SPRG[4-7]</td>
<td>-</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>268</td>
<td>01000 01100</td>
<td>TB</td>
<td>-</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>269</td>
<td>01000 01101</td>
<td>TBU</td>
<td>-</td>
<td>32</td>
<td>B</td>
</tr>
<tr>
<td>272-275</td>
<td>01000 100xx</td>
<td>SPRG[0-3]</td>
<td>yes</td>
<td>64</td>
<td>B</td>
</tr>
<tr>
<td>276-279</td>
<td>01000 101xx</td>
<td>SPRG[4-7]</td>
<td>yes</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>282</td>
<td>01000 11010</td>
<td>EAR</td>
<td>hypv^4</td>
<td>32</td>
<td>EC</td>
</tr>
<tr>
<td>283</td>
<td>01000 11011</td>
<td>CIR</td>
<td>-</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>284</td>
<td>01000 11100</td>
<td>TBL</td>
<td>hypv^4</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>285</td>
<td>01000 11101</td>
<td>TBU</td>
<td>hypv^4</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>286</td>
<td>01000 11110</td>
<td>TBU40</td>
<td>hypv^4</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>287</td>
<td>01000 11111</td>
<td>PVR</td>
<td>-</td>
<td>32</td>
<td>B</td>
</tr>
<tr>
<td>304</td>
<td>01001 10000</td>
<td>HSPRG0</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>305</td>
<td>01001 10001</td>
<td>HSPRG1</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>306</td>
<td>01001 10010</td>
<td>HDSISR</td>
<td>hypv^3</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>307</td>
<td>01001 10011</td>
<td>HDAR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>308</td>
<td>01001 10100</td>
<td>SPURR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>309</td>
<td>01001 10101</td>
<td>DBCR0</td>
<td>hypv^3</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>310</td>
<td>01001 10102</td>
<td>PURR</td>
<td>hypv^3</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>311</td>
<td>01001 10103</td>
<td>DBCR1</td>
<td>hypv^3</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>312</td>
<td>01001 10104</td>
<td>HDEC</td>
<td>hypv^3</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>313</td>
<td>01001 10105</td>
<td>DBCR2</td>
<td>hypv^3</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>314</td>
<td>01001 10106</td>
<td>MSRPR</td>
<td>hypv^3</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>315</td>
<td>01001 10107</td>
<td>RMROR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>316</td>
<td>01001 10108</td>
<td>IAC1</td>
<td>hypv^3</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>317</td>
<td>01001 10109</td>
<td>IAC2</td>
<td>hypv^3</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>318</td>
<td>01001 10110</td>
<td>IAC3</td>
<td>hypv^3</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>319</td>
<td>01001 10111</td>
<td>IAC4</td>
<td>hypv^3</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>320</td>
<td>01001 10112</td>
<td>HSMR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>321</td>
<td>01001 10113</td>
<td>IAC5</td>
<td>hypv^3</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>322</td>
<td>01001 10114</td>
<td>LPCR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>323</td>
<td>01001 10115</td>
<td>LPIDR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>324</td>
<td>01001 10116</td>
<td>TSR</td>
<td>yes^5,13</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>325</td>
<td>01001 10117</td>
<td>HMER</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>326</td>
<td>01001 10118</td>
<td>LMEE</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>327</td>
<td>01001 10119</td>
<td>PCR</td>
<td>hypv^3</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>328</td>
<td>01001 10120</td>
<td>LPIDR</td>
<td>hypv^3</td>
<td>32</td>
<td>E HV</td>
</tr>
<tr>
<td>329</td>
<td>01001 10121</td>
<td>HEIR</td>
<td>hypv^3</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>330</td>
<td>01001 10122</td>
<td>MA55</td>
<td>hypv^3</td>
<td>32</td>
<td>E HV</td>
</tr>
<tr>
<td>331</td>
<td>01001 10123</td>
<td>TCR</td>
<td>yes^13</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>332</td>
<td>01001 10124</td>
<td>MAS8</td>
<td>hypv^3</td>
<td>32</td>
<td>E HV</td>
</tr>
</tbody>
</table>

Figure 2. SPR Numbers (Sheet 2 of 5)
<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length (bits)</th>
<th>Cat&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>342</td>
<td>01010 10110</td>
<td>LRATCFG</td>
<td>-</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>E.HV.LRAT</td>
</tr>
<tr>
<td>343</td>
<td>01010 10111</td>
<td>LRATPS</td>
<td>-</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>E.HV.LRAT</td>
</tr>
<tr>
<td>344-347</td>
<td>01010 1100x</td>
<td>TLB[0-3]PS</td>
<td>-</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>E.HV</td>
</tr>
<tr>
<td>348</td>
<td>01010 11100</td>
<td>MAS7IMAS6</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>E.HV, 64</td>
</tr>
<tr>
<td>349</td>
<td>01010 11101</td>
<td>MAS8IMAS1</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>E.HV, 64</td>
</tr>
<tr>
<td>349</td>
<td>01010 11100</td>
<td>AMOR</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;3&lt;/sup&gt;</td>
<td>S</td>
</tr>
<tr>
<td>350</td>
<td>01010 11110</td>
<td>EPTCFG</td>
<td>hypv&lt;sup&gt;9&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;9&lt;/sup&gt;</td>
<td>E.PT</td>
</tr>
<tr>
<td>368-371</td>
<td>01011 100xx</td>
<td>GSPRG0-3</td>
<td>yes yes</td>
<td>64</td>
<td>E.HV</td>
</tr>
<tr>
<td>372</td>
<td>01011 10100</td>
<td>MAS7IMAS3</td>
<td>yes yes</td>
<td>64</td>
<td>E: 64</td>
</tr>
<tr>
<td>373</td>
<td>01011 10110</td>
<td>MAS0IMAS1</td>
<td>yes yes</td>
<td>64</td>
<td>E: 64</td>
</tr>
<tr>
<td>374</td>
<td>01011 10111</td>
<td>GDEC</td>
<td>yes yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>375</td>
<td>01011 10111</td>
<td>GTCR</td>
<td>yes yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>376</td>
<td>01011 11000</td>
<td>GTSR</td>
<td>yes yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>378</td>
<td>01011 11010</td>
<td>GSRR0</td>
<td>yes yes</td>
<td>64</td>
<td>E.HV</td>
</tr>
<tr>
<td>379</td>
<td>01011 11011</td>
<td>GSRR1</td>
<td>yes yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>380</td>
<td>01011 11100</td>
<td>GEPR</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>383</td>
<td>01011 11111</td>
<td>GESR</td>
<td>yes yes</td>
<td>32</td>
<td>E.HV</td>
</tr>
<tr>
<td>400-415</td>
<td>01100 1xxxx</td>
<td>IVOR[0-15]</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>432-435</td>
<td>01101 100xx</td>
<td>IVOR38-41</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>436</td>
<td>01101 10100</td>
<td>IVOR42</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>437</td>
<td>01101 10101</td>
<td>TENS</td>
<td>-</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>438</td>
<td>01101 10110</td>
<td>TENS</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>439</td>
<td>01101 10111</td>
<td>TENC</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>440-441</td>
<td>01101 1100x</td>
<td>GIVOR2-3</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>442</td>
<td>01101 11010</td>
<td>GIVOR4</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>443</td>
<td>01101 11011</td>
<td>GIVOR8</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>444</td>
<td>01101 11100</td>
<td>GIVOR13</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>445</td>
<td>01101 11101</td>
<td>GIVOR14</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>446</td>
<td>01101 11110</td>
<td>TIR</td>
<td>-</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>446</td>
<td>01101 11111</td>
<td>TIR</td>
<td>-</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>447</td>
<td>01101 11111</td>
<td>GIVPR</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>464</td>
<td>01110 10000</td>
<td>GIVOR35</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>474</td>
<td>11010 01110</td>
<td>GIVOR10</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>475</td>
<td>11011 01110</td>
<td>GIVOR11</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>476</td>
<td>11100 01110</td>
<td>GIVOR12</td>
<td>hypv&lt;sup&gt;4&lt;/sup&gt;</td>
<td>yes</td>
<td>32</td>
</tr>
<tr>
<td>512</td>
<td>10000 00000</td>
<td>SPEFSR</td>
<td>no</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>526</td>
<td>10000 01110</td>
<td>ATB/ATBL</td>
<td>-</td>
<td>no</td>
<td>64</td>
</tr>
<tr>
<td>527</td>
<td>10000 01111</td>
<td>ATBU</td>
<td>-</td>
<td>no</td>
<td>32</td>
</tr>
<tr>
<td>528</td>
<td>10000 10000</td>
<td>IVOR32</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>529</td>
<td>10000 10001</td>
<td>IVOR33</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>530</td>
<td>10000 10010</td>
<td>IVOR34</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>531</td>
<td>10000 10011</td>
<td>IVOR35</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>532</td>
<td>10000 10100</td>
<td>IVOR36</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>533</td>
<td>10000 10101</td>
<td>IVOR37</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>570</td>
<td>10001 11010</td>
<td>MCSRR0</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>571</td>
<td>10001 11110</td>
<td>MCSRR1</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>32</td>
</tr>
<tr>
<td>572</td>
<td>10001 11111</td>
<td>MCSR</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>574</td>
<td>10001 11110</td>
<td>DSRR0</td>
<td>yes yes</td>
<td>64</td>
<td>E.ED</td>
</tr>
<tr>
<td>575</td>
<td>10001 11111</td>
<td>DSRR1</td>
<td>yes yes</td>
<td>32</td>
<td>E.ED</td>
</tr>
<tr>
<td>604</td>
<td>10010 11100</td>
<td>SPRG8</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>64</td>
</tr>
<tr>
<td>605</td>
<td>10010 11110</td>
<td>SPRG9</td>
<td>yes yes</td>
<td>64</td>
<td>E.ED</td>
</tr>
<tr>
<td>624</td>
<td>10011 10000</td>
<td>MAS0</td>
<td>yes yes</td>
<td>32</td>
<td>E</td>
</tr>
</tbody>
</table>

Figure 2. SPR Numbers (Sheet 3 of 5)
<table>
<thead>
<tr>
<th>decimal</th>
<th>SPR1&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Register Name</th>
<th>Privileged</th>
<th>Length</th>
<th>Cat&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>625</td>
<td>10011 10001</td>
<td>MAS1</td>
<td>yes</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>626</td>
<td>10011 10010</td>
<td>MAS2</td>
<td>yes</td>
<td>64</td>
<td>E</td>
</tr>
<tr>
<td>627</td>
<td>10011 10011</td>
<td>MAS3</td>
<td>yes</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>628</td>
<td>10011 10100</td>
<td>MAS4</td>
<td>yes</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>630</td>
<td>10011 10110</td>
<td>MAS6</td>
<td>yes</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>631</td>
<td>10011 10111</td>
<td>MAS2U</td>
<td>yes</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>688-691</td>
<td>10101 100xx</td>
<td>TLB[0-3]CFG</td>
<td>-</td>
<td>32</td>
<td>E</td>
</tr>
<tr>
<td>702</td>
<td>10101 11110</td>
<td>EPR</td>
<td>-</td>
<td>32</td>
<td>EXP</td>
</tr>
<tr>
<td>768</td>
<td>11000 00000</td>
<td>SIER</td>
<td>-</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>769</td>
<td>11000 00001</td>
<td>MMCRA</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>770</td>
<td>11000 00010</td>
<td>MMCRA</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>771</td>
<td>11000 00011</td>
<td>PMC1</td>
<td>no</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>772</td>
<td>11000 00100</td>
<td>PMC2</td>
<td>no</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>773</td>
<td>11000 00101</td>
<td>PMC3</td>
<td>no</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>774</td>
<td>11000 00110</td>
<td>PMC4</td>
<td>no</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>775</td>
<td>11000 00111</td>
<td>PMC5</td>
<td>no</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>776</td>
<td>11000 01000</td>
<td>PMC6</td>
<td>no</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>779</td>
<td>11000 01010</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>780</td>
<td>11000 01011</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>781</td>
<td>11000 01100</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>782</td>
<td>11000 01101</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>784</td>
<td>11000 10000</td>
<td>SIAR</td>
<td>-</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>785</td>
<td>11000 10001</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>786</td>
<td>11000 10010</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>787</td>
<td>11000 10011</td>
<td>PMC1</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>788</td>
<td>11000 10100</td>
<td>PMC2</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>789</td>
<td>11000 10101</td>
<td>PMC3</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>790</td>
<td>11000 10110</td>
<td>PMC4</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>791</td>
<td>11000 10111</td>
<td>PMC5</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>792</td>
<td>11000 11000</td>
<td>PMC6</td>
<td>yes</td>
<td>32</td>
<td>S</td>
</tr>
<tr>
<td>795</td>
<td>11000 11010</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>796</td>
<td>11000 11011</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>797</td>
<td>11000 11100</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>798</td>
<td>11000 11101</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>799</td>
<td>11000 11110</td>
<td>MMCRA</td>
<td>yes</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>800</td>
<td>11001 00000</td>
<td>BESCRS</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>801</td>
<td>11001 00001</td>
<td>BESCRSU</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>802</td>
<td>11001 00010</td>
<td>BESCRR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>803</td>
<td>11001 00011</td>
<td>BESCRRU</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>804</td>
<td>11001 00100</td>
<td>EBBHR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>805</td>
<td>11001 00101</td>
<td>EBBRR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>806</td>
<td>11001 00110</td>
<td>BESCR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>808</td>
<td>11001 01000</td>
<td>reserved&lt;sup&gt;15&lt;/sup&gt;</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>809</td>
<td>11001 01010</td>
<td>reserved&lt;sup&gt;15&lt;/sup&gt;</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>810</td>
<td>11001 01011</td>
<td>reserved&lt;sup&gt;15&lt;/sup&gt;</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>811</td>
<td>11001 01100</td>
<td>reserved&lt;sup&gt;15&lt;/sup&gt;</td>
<td>no</td>
<td>no</td>
<td>na</td>
</tr>
<tr>
<td>815</td>
<td>11001 01111</td>
<td>TAR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>848</td>
<td>11010 10000</td>
<td>IC</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>849</td>
<td>11010 10001</td>
<td>VTB</td>
<td>hypv&lt;sup&gt;12&lt;/sup&gt;</td>
<td>yes</td>
<td>64</td>
</tr>
<tr>
<td>896</td>
<td>11100 00000</td>
<td>PPR</td>
<td>no</td>
<td>64</td>
<td>S</td>
</tr>
<tr>
<td>898</td>
<td>11100 00010</td>
<td>PPR32</td>
<td>no</td>
<td>32</td>
<td>B</td>
</tr>
<tr>
<td>924</td>
<td>11100 11100</td>
<td>DCDBTRL</td>
<td>-</td>
<td>32</td>
<td>E.CD</td>
</tr>
<tr>
<td>925</td>
<td>11100 11101</td>
<td>DCDBTRH</td>
<td>-</td>
<td>32</td>
<td>E.CD</td>
</tr>
<tr>
<td>926</td>
<td>11100 11110</td>
<td>ICDBTRL</td>
<td>-</td>
<td>32</td>
<td>E.CD</td>
</tr>
<tr>
<td>927</td>
<td>11100 11111</td>
<td>ICDBTRH</td>
<td>-</td>
<td>32</td>
<td>E.CD</td>
</tr>
</tbody>
</table>

Figure 2. SPR Numbers (Sheet 4 of 5)
This register is not defined for this instruction.
1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 See Section 1.3.5 of Book I. If multiple categories are listed separated by a semicolon, all the listed categories must be implemented in order for the other columns of the line to apply. A comma separates two alternatives, and takes precedence over a semicolon; e.g., the EPCR (E.HV,E;64) must be implemented if either (a) category E.HV is implemented or (b) the processor is an Embedded processor that implements the 64-Bit category.
3 This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-S or Chapter 2 of Book III-E as appropriate).
4 <S>This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-S).
5 This register cannot be directly written. Instead, bits in the register corresponding to 1 bits in (RS) can be cleared using `mtspr SPR,RS`.
6 The register can be written by the `dcread` instruction.
7 The register can be written by the `icread` instruction.
8 This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using `mtspr SPR,RS`.
9 The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the `mtspr` instruction description in Book III-S.
10 The value specified in register RS may be ANDed with the contents of the AMOR before being placed into the UAMOR; see the `mtspr` instruction description in Book III-S.
11 The register is Category: Phased-in.
12 If the Embedded.Hypervisor category is supported, this register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E). Otherwise the register is privileged for Embedded.
13 If the Embedded.Hypervisor category is supported, this register is a hypervisor resource and can be accessed by this instruction only in hypervisor state, and guest references to the register are redirected to the corresponding guest register (see Chapter 2 of Book III-E). Otherwise the register is privileged.
14 MMCR0_PMCC controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 9.4 for details.
15 Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.

SPR numbers 777-778, 783, 793-794, and 799 are reserved for the Performance Monitor. All other SPR numbers that are not shown above and are not implementation-specific are reserved.

Figure 2. SPR Numbers (Sheet 5 of 5)
Appendix D. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

1, 5, 6

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix F of Book Appendices. All unused extended opcodes are illegal.

4, 19, 30, 31, 56, 5, 58, 59, 60, 62, 63

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.
Appendix E. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the Power ISA.

The following types of instruction are included in this class.

1. The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction; see Section 1.7.2, “Illegal Instruction Class” on page 21) and the extended opcode shown below.

   256   Service Processor “Attention”

2. Instructions for the POWER Architecture that have not been included in the Power ISA. These are listed in Section A.31, “Discontinued Opcodes” and Section A.33.1, “Discontinued Opcodes”.

3. Implementation-specific instructions used to conform to the Power ISA specification.

4. Any other implementation-dependent instructions that are not defined in the Power ISA.
Appendix F. Opcode Maps

This appendix contains tables showing the opcodes and extended opcodes.

For the primary opcode table (Table 1 on page 1372), each cell is in the following format.

<table>
<thead>
<tr>
<th>Opcode in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
</tr>
<tr>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

  Instruction
  Mnemonic

The category abbreviations are shown on Section 1.3.5 of Book I. However, the categories "Phased-In", "Phased-Out", and floating-point "Record" are not listed in the opcode tables.

The extended opcode tables show the extended opcode in decimal, the instruction mnemonic, the category, and the instruction format. These tables appear in order of primary opcode within three groups. The first group consists of the primary opcodes that have small extended opcode fields (2-4 bits), namely 30, 58, and 62. The second group consists of primary opcodes that have 11-bit extended opcode fields. The third group consists of primary opcodes that have 10-bit extended opcode fields. The tables for the second and third groups are rotated.

In the extended opcode tables several special markings are used.

- A prime (') following an instruction mnemonic denotes an additional cell, after the lowest-numbered one, used by the instruction. For example, `subfc` occupies cells 8 and 520 of primary opcode 31, with the former corresponding to OE=0 and the latter to OE=1. Similarly, `sradi` occupies cells 826 and 827, with the former corresponding to sh5=0 and the latter to sh5=1 (the 9-bit extended opcode 413, shown on page 100, excludes the sh5 bit).

- Two vertical bars (||) are used instead of primed mnemonics when an instruction occupies an entire column of a table. The instruction mnemonic is repeated in the last cell of the column.

- For primary opcode 31, an asterisk (*) in a cell that would otherwise be empty means that the cell is reserved because it is "overlaid", by a fixed-point or `Storage Access` instruction having only a primary opcode, by an instruction having an extended opcode in primary opcode 30, 58, or 62, or by a potential instruction in any of the categories just mentioned. The overlaying instruction, if any, is also shown. A cell thus reserved should not be assigned to an instruction having primary opcode 31. (The overlaying is a consequence of opcode decoding for fixed-point instructions: the primary opcode, and the extended opcode if any, are mapped internally to a 10-bit "compressed opcode" for ease of subsequent decoding on some implementations that complied with previous versions of the architecture.)

- Parentheses around the opcode or extended opcode mean that the instruction was defined in earlier versions of the Power ISA but is no longer defined in the Power ISA.

- Curly brackets around the opcode or extended opcode mean that the instruction will be defined in future versions of the Power ISA.

- `long` is used as filler for mnemonics that are longer than a table cell.

An empty cell, a cell containing only an asterisk, or a cell in which the opcode or extended opcode is parenthesized, corresponds to an illegal instruction.

The instruction consisting entirely of binary 0s causes the system illegal instruction error handler to be invoked for all members of the POWER family, and this is likely to remain true in future models (it is guaranteed in the Power ISA). An instruction having primary opcode 0 but not consisting entirely of binary 0s is reserved except for the following extended opcode (instruction bits 21:30).

256 Service Processor “Attention” (Power ISA only)
<table>
<thead>
<tr>
<th>Primary opcodes</th>
<th>Illegal, Reserved</th>
<th>tdi</th>
<th>twi</th>
<th>multi</th>
<th>Trap Doubleword Immediate</th>
<th>Trap Word Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>See primary opcode 0 extensions on page 1371</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>See Table 8 and Table 7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>subf</td>
<td>08</td>
<td>09</td>
<td>10</td>
<td>Subtract From Immediate Carrying</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>Compare Logical Immediate</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>addic</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>Add Immediate Carrying</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>Add Immediate Carrying and Record</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>bc</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>Branch Conditional</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>System Call</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>rwiimi</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>Rotate Left Word Imm. then Mask Insert</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>Rotate Left Word Imm. then AND with Mask</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>ori</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>OR Immediate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>OR Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>andi.</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>AND Immediate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>AND Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>lwz</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>Load Half and Zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>Load Half and Zero with Update</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>stw</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>Load Half Algebraic</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>Load Half Algebraic with Update</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>lhz</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>Load Floating-Point Single</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>Load Floating-Point Single with Update</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>sth</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>Load Floating-Point Double</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>Load Floating-Point Double with Update</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>lfs</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>Store Floating-Point Single</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>Store Floating-Point Single with Update</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>stfs</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>Store Floating-Point Double</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>Store Floating-Point Double with Update</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>lq</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>Load Quadword</td>
<td></td>
</tr>
<tr>
<td>LSQ</td>
<td></td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>See Table 3 on page 1374</td>
<td></td>
</tr>
<tr>
<td>DQ</td>
<td></td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>See Table 4 on page 1374</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>See Table 16 on page 1389</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1: Primary opcodes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Illegal, Reserved</td>
</tr>
<tr>
<td>01</td>
<td>tdi</td>
</tr>
<tr>
<td>02</td>
<td>twi</td>
</tr>
<tr>
<td>03</td>
<td>multi</td>
</tr>
<tr>
<td>04</td>
<td>Vector, LMA, SP</td>
</tr>
<tr>
<td>05</td>
<td>tdi</td>
</tr>
<tr>
<td>06</td>
<td>twi</td>
</tr>
<tr>
<td>07</td>
<td>multi</td>
</tr>
<tr>
<td>08</td>
<td>subf</td>
</tr>
<tr>
<td>09</td>
<td>cmpli</td>
</tr>
<tr>
<td>10</td>
<td>cmpli</td>
</tr>
<tr>
<td>11</td>
<td>cmpli</td>
</tr>
<tr>
<td>12</td>
<td>addic.</td>
</tr>
<tr>
<td>13</td>
<td>addic.</td>
</tr>
<tr>
<td>14</td>
<td>addi</td>
</tr>
<tr>
<td>15</td>
<td>addis</td>
</tr>
<tr>
<td>16</td>
<td>bc</td>
</tr>
<tr>
<td>17</td>
<td>sc</td>
</tr>
<tr>
<td>18</td>
<td>b</td>
</tr>
<tr>
<td>19</td>
<td>CR ops, etc.</td>
</tr>
<tr>
<td>20</td>
<td>rwiimi</td>
</tr>
<tr>
<td>21</td>
<td>rlwinm</td>
</tr>
<tr>
<td>22</td>
<td>rlwinm</td>
</tr>
<tr>
<td>23</td>
<td>rlwinm</td>
</tr>
<tr>
<td>24</td>
<td>ori</td>
</tr>
<tr>
<td>25</td>
<td>oris</td>
</tr>
<tr>
<td>26</td>
<td>xori</td>
</tr>
<tr>
<td>27</td>
<td>xoris</td>
</tr>
<tr>
<td>28</td>
<td>andis.</td>
</tr>
<tr>
<td>29</td>
<td>andis.</td>
</tr>
<tr>
<td>30</td>
<td>FX Dwd Rot</td>
</tr>
<tr>
<td>31</td>
<td>FX Extended Ops</td>
</tr>
<tr>
<td>32</td>
<td>lwz</td>
</tr>
<tr>
<td>33</td>
<td>lwzu</td>
</tr>
<tr>
<td>34</td>
<td>lbz</td>
</tr>
<tr>
<td>35</td>
<td>lbzu</td>
</tr>
<tr>
<td>36</td>
<td>stw</td>
</tr>
<tr>
<td>37</td>
<td>stwu</td>
</tr>
<tr>
<td>38</td>
<td>stb</td>
</tr>
<tr>
<td>39</td>
<td>stbu</td>
</tr>
<tr>
<td>40</td>
<td>lhz</td>
</tr>
<tr>
<td>41</td>
<td>lhzu</td>
</tr>
<tr>
<td>42</td>
<td>lh</td>
</tr>
<tr>
<td>43</td>
<td>lhau</td>
</tr>
<tr>
<td>44</td>
<td>sth</td>
</tr>
<tr>
<td>45</td>
<td>sthu</td>
</tr>
<tr>
<td>46</td>
<td>lm</td>
</tr>
<tr>
<td>47</td>
<td>stmw</td>
</tr>
<tr>
<td>48</td>
<td>lfs</td>
</tr>
<tr>
<td>49</td>
<td>lfsu</td>
</tr>
<tr>
<td>50</td>
<td>lfd</td>
</tr>
<tr>
<td>51</td>
<td>lfdu</td>
</tr>
<tr>
<td>52</td>
<td>stfs</td>
</tr>
<tr>
<td>53</td>
<td>stfsu</td>
</tr>
<tr>
<td>54</td>
<td>std</td>
</tr>
<tr>
<td>55</td>
<td>stdu</td>
</tr>
<tr>
<td>56</td>
<td>lq</td>
</tr>
<tr>
<td>57</td>
<td>FX DS-form Loads</td>
</tr>
<tr>
<td>58</td>
<td>FP Single &amp; DFP Ops</td>
</tr>
<tr>
<td>59</td>
<td>Load Quadword</td>
</tr>
</tbody>
</table>

Table 1: Primary opcodes

<table>
<thead>
<tr>
<th>60</th>
<th>3C</th>
<th>61</th>
<th>3D</th>
<th>62</th>
<th>3E</th>
<th>63</th>
<th>3F</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSX Extended Ops</td>
<td>stdp</td>
<td>FX DS-form Stores</td>
<td>FP Double &amp;DFP Ops</td>
<td>Store Floating-Point Double Pair</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP</td>
<td>DS</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 6 on page 1374
See Table 17 on page 1391
See Table 18 on page 1393
## Table 2: Extended opcodes for primary opcode 30

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Extended opcodes for primary opcode 30 (instruction bits 27:30)

- **rldicl**: 64 MD
- **rldicl’**: 64 MD
- **rldi**: 64 MD
- **rldi’**: 64 MD
- **rldicr**: 64 MD
- **rldicr’**: 64 MD

## Table 3: Extended opcodes for primary opcode 57

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended opcodes for primary opcode 57 (instruction bits 30:31)

- **lfdp**: FP DS
- **ld**: 64 DS
- **ldu**: 64 DS
- **lwa**: 64 DS

## Table 4: Extended opcodes for primary opcode 58

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended opcodes for primary opcode 58 (instruction bits 30:31)

- **std**: 64 DS
- **stdu**: 64 DS
- **stq**: LSQ DS

## Table 5: Extended opcodes for primary opcode 61

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended opcodes for primary opcode 61 (instruction bits 30:31)

- **stfdp**: FP DS
- **std**: 64 DS

## Table 6: Extended opcodes for primary opcode 62

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended opcodes for primary opcode 62 (instruction bits 30:31)

- **stq**: LSQ DS
Table 7: (Left) Extended opcodes for primary opcode 4 [Category: SP*] (instruction bits 21:31)

<table>
<thead>
<tr>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>000100</th>
<th>000101</th>
<th>000110</th>
<th>000111</th>
<th>001000</th>
<th>001001</th>
<th>001010</th>
<th>001011</th>
<th>001100</th>
<th>001101</th>
<th>001110</th>
<th>001111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>00000</td>
<td>01000</td>
<td>00001</td>
<td>01001</td>
<td>00010</td>
<td>01010</td>
<td>00011</td>
<td>01011</td>
<td>00100</td>
<td>01100</td>
<td>00101</td>
<td>01101</td>
<td>00110</td>
<td>01110</td>
<td>00111</td>
<td>01111</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>529</td>
<td>530</td>
<td>531</td>
<td>532</td>
<td>533</td>
<td>534</td>
<td>535</td>
<td>536</td>
<td>537</td>
<td>538</td>
<td>539</td>
<td>540</td>
<td>541</td>
<td>542</td>
<td>543</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td>evand</td>
<td></td>
</tr>
<tr>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td>evxc</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td>sp</td>
<td></td>
</tr>
<tr>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td>evor</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>544 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>545 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>546 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>547 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>548 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>549 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>736 efsladd sp. MEVX sp. MEVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>737 efslsub sp. MEVX sp. MEVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>800 overload SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1059 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1127 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1128 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1132 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1133 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>1320 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1321 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1322 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1323 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1446 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1449 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1451 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1452 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1453 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1455 long SP EVX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7 (Right) Extended opcodes for primary opcode 4 [Category: SP*] (instruction bits 21:31)

<table>
<thead>
<tr>
<th>111111</th>
<th>111110</th>
<th>111101</th>
<th>111100</th>
<th>111011</th>
<th>111010</th>
<th>111001</th>
<th>111000</th>
<th>110111</th>
<th>110110</th>
<th>110101</th>
<th>110100</th>
<th>110011</th>
<th>110010</th>
<th>110001</th>
<th>110000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 8: (Left) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 20:16</th>
<th>Bits 15:12</th>
<th>Bits 11:8</th>
<th>Bits 7:4</th>
<th>Bits 3:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1101011</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1111011</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>1110010</td>
<td>1110011</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1379  Power ISA™ - Book Appendices
Table 8 (Left-Center) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>16</th>
<th>mulhhuw</th>
<th>17</th>
<th>mulhhwu</th>
<th>18</th>
<th>mulhhwu</th>
<th>19</th>
<th>mulhhwu</th>
<th>20</th>
<th>mulhhwu</th>
<th>21</th>
<th>mulhhwu</th>
<th>22</th>
<th>mulhhwu</th>
<th>23</th>
<th>mulhhwu</th>
<th>24</th>
<th>mulhhwu</th>
<th>25</th>
<th>mulhhwu</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00000</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>00001</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>00010</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>00011</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>00100</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>00101</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>00110</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>00111</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>01000</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>01001</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>01010</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>01100</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>01101</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>01110</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>10000</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>10001</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>10010</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>10011</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>10100</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>10101</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>10110</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>10111</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>11000</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>11001</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>11010</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>11011</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>11110</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>11111</td>
<td>mulhhuw</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td>mulhhwu</td>
<td>LX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>32</td>
<td>vmaddbst</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td>32</td>
<td>VMAA</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
<td>VMADDST</td>
</tr>
<tr>
<td></td>
<td>11111</td>
<td>11110</td>
<td>11101</td>
<td>11100</td>
<td>11011</td>
<td>11010</td>
<td>11001</td>
<td>11000</td>
<td>10111</td>
<td>10110</td>
<td>10101</td>
<td>10100</td>
<td>10011</td>
<td>10010</td>
<td>10001</td>
<td>10000</td>
<td>01111</td>
<td>01110</td>
<td>01101</td>
<td>01100</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9: (Left) Extended opcodes for primary opcode 19 (instruction bits 21:30)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>mcrf</th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</th>
<th>01000</th>
<th>01001</th>
<th>01010</th>
<th>01011</th>
<th>01100</th>
<th>01101</th>
<th>01110</th>
<th>01111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td>00111</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
</tr>
<tr>
<td>00111</td>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td>00000</td>
<td>00001</td>
<td>00010</td>
<td>00011</td>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
<td></td>
</tr>
</tbody>
</table>
Table 9. (Right) Extended opcodes for primary opcode 19 (instruction bits 21:30)

<table>
<thead>
<tr>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</th>
<th>01000</th>
<th>01001</th>
<th>01010</th>
<th>01011</th>
<th>01100</th>
<th>01101</th>
<th>01110</th>
<th>01111</th>
<th>10000</th>
<th>10001</th>
<th>10010</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 18h | rth | XL |
| 19h | rfi | XL |
| 50h | rle | XL |
| 51h | rlc | XL |
| 80h | rsvc | XL |
| 146h | rthbb | XL |
| 150h | isync | XL |
| 274h | brcd | XL |
| 402h | doze | XL |
| 434h | nap | XL |
| 456h | sleep | XL |
| 488h | rynw | XL |
| 528h | bcotr | XL |
| 550h | btyat | XL |

[Table continued on the next page]
| Table 10: Left Extended opcodes for primary opcode 31 (instruction bits 21:30) |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|      | 00000 | 00001 | 00010 | 00011 | 00100 | 00101 | 00110 | 00111 |
| 00000 | 0 comp B X | 32 rcmp B X | 31 31 Resd VLE | 68 rf 64 X | 71 lvex V X | 75 mulhd 64 XO | 74 adddq BCVA 64 XO | 55 lpmul V SX X | 75 lmpd B SX PC X |
| 00001 | 32 rcmp B X | 31 31 Resd VLE | 68 rf 64 X | 71 lvex V X | 75 mulhd 64 XO | 74 adddq BCVA 64 XO | 55 lpmul V SX X | 75 lmpd B SX PC X |
| 00010 | 129 31 Resd VLE | 121 write E X | 120 rdoc casl ECL X | 103 ssbus V X | 104 neg B XO | 103 ssbus V X | 104 neg B XO | 103 ssbus V X | 104 neg B XO |
| 00011 | 193 31 Resd VLE | 181 stdex V X | 180 stdex V X | 180 stdex V X | 200 stdex V X | 200 stdex V X | 200 stdex V X | 200 stdex V X | 200 stdex V X |
| 00100 | 237 31 Resd VLE | 229 mfpex E DC X | 228 rdoc E PD X | 203 stdex V X | 202 adde B XO | 202 adde B XO | 202 adde B XO | 202 adde B XO | 202 adde B XO |
| 00101 | 291 31 Resd VLE | 269 mpex E DC X | 268 rdoc E PD X | 203 stdex V X | 202 adde B XO | 202 adde B XO | 202 adde B XO | 202 adde B XO | 202 adde B XO |
| 00101 | 391 31 Resd VLE | 369 mfpex E DC X | 368 rdoc E PD X | 203 stdex V X | 202 adde B XO | 202 adde B XO | 202 adde B XO | 202 adde B XO | 202 adde B XO |
| 00110 | 417 31 Resd VLE | 397 stdex E DC X | 396 stdex E DC X | 393 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO |
| 00111 | 449 31 Resd VLE | 401 stdex E DC X | 400 stdex E DC X | 397 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO | 395 divdeu 64 XO |
| 10000 | 512 microx E X | 515 ldex DS X | 514 stdex V X | 513 stdex V X | 512 stdex V X | 512 stdex V X | 512 stdex V X | 512 stdex V X | 512 stdex V X |
| 10001 | 518 stdex DS X | 517 stdex V X | 516 stdex V X | 515 stdex V X | 514 stdex V X | 513 stdex V X | 512 stdex V X | 512 stdex V X | 512 stdex V X |
| 10010 | 519 stdex DS X | 518 stdex V X | 517 stdex V X | 516 stdex V X | 515 stdex V X | 514 stdex V X | 513 stdex V X | 512 stdex V X | 512 stdex V X |
| 10100 | 643 stdex DS X | 642 stdex V X | 641 stdex V X | 640 stdex V X | 639 stdex V X | 638 stdex V X | 637 stdex V X | 636 stdex V X | 635 stdex V X |
| 10101 | 679 stdex DS X | 678 stdex V X | 677 stdex V X | 676 stdex V X | 675 stdex V X | 674 stdex V X | 673 stdex V X | 672 stdex V X | 671 stdex V X |
| 10110 | 747 stdex DS X | 746 stdex V X | 745 stdex V X | 744 stdex V X | 743 stdex V X | 742 stdex V X | 741 stdex V X | 740 stdex V X | 739 stdex V X |
| 11000 | 715 stdex DS X | 714 stdex V X | 713 stdex V X | 712 stdex V X | 711 stdex V X | 710 stdex V X | 709 stdex V X | 708 stdex V X | 707 stdex V X |
| 11001 | 833 stdex DS X | 832 stdex V X | 831 stdex V X | 830 stdex V X | 829 stdex V X | 828 stdex V X | 827 stdex V X | 826 stdex V X | 825 stdex V X |
| 11010 | 993 stdex DS X | 992 stdex V X | 991 stdex V X | 990 stdex V X | 989 stdex V X | 988 stdex V X | 987 stdex V X | 986 stdex V X | 985 stdex V X |
| 11100 | 131 stdex DS X | 130 stdex V X | 129 stdex V X | 128 stdex V X | 127 stdex V X | 126 stdex V X | 125 stdex V X | 124 stdex V X | 123 stdex V X |
| 11111 | 199 stdex DS X | 198 stdex V X | 197 stdex V X | 196 stdex V X | 195 stdex V X | 194 stdex V X | 193 stdex V X | 192 stdex V X | 191 stdex V X |
Table 10. (Right) Extended opcodes for primary opcode 31 (instruction bits 21:30)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16: Response & VFP
15: Memory type & VFP
14: Memory type & VFP
13: Memory type & VFP
12: Memory type & VFP
11: Memory type & VFP
10: Memory type & VFP
9: Memory type & VFP
8: Memory type & VFP
7: Memory type & VFP
6: Memory type & VFP
5: Memory type & VFP
4: Memory type & VFP
3: Memory type & VFP
2: Memory type & VFP
1: Memory type & VFP
0: Memory type & VFP
### Table 11: Opcode: 31, Extended Opcode: 30

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11110</td>
<td>0</td>
<td>rldicl*</td>
</tr>
</tbody>
</table>

### Table 12: Opcode: 31, Extended Opcode: 62

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11110</td>
<td>0</td>
<td>rldicl*</td>
</tr>
</tbody>
</table>

### Table 13: Opcode: 31, Extended Opcode: 854

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110</td>
<td>11010</td>
<td>0</td>
<td>eieio</td>
</tr>
<tr>
<td>10110</td>
<td>11010</td>
<td>1</td>
<td>mbar</td>
</tr>
</tbody>
</table>

### Table 14: Opcode: 31, Extended Opcode: 159

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111</td>
<td>00100</td>
<td>1</td>
<td>eieio</td>
</tr>
<tr>
<td>11111</td>
<td>00101</td>
<td>2</td>
<td>mbar</td>
</tr>
<tr>
<td>11111</td>
<td>00110</td>
<td>3</td>
<td>sthepx</td>
</tr>
<tr>
<td>11111</td>
<td>00100</td>
<td>4</td>
<td>eieio</td>
</tr>
<tr>
<td>11111</td>
<td>00101</td>
<td>5</td>
<td>mbar</td>
</tr>
<tr>
<td>11111</td>
<td>00110</td>
<td>6</td>
<td>sthepx</td>
</tr>
<tr>
<td>11111</td>
<td>01000</td>
<td>7</td>
<td>eieio</td>
</tr>
<tr>
<td>11111</td>
<td>01001</td>
<td>8</td>
<td>mbar</td>
</tr>
<tr>
<td>11111</td>
<td>01010</td>
<td>9</td>
<td>sthepx</td>
</tr>
<tr>
<td>11111</td>
<td>01011</td>
<td>10</td>
<td>eieio</td>
</tr>
<tr>
<td>11111</td>
<td>01100</td>
<td>11</td>
<td>mbar</td>
</tr>
<tr>
<td>11111</td>
<td>01101</td>
<td>12</td>
<td>sthepx</td>
</tr>
<tr>
<td>11111</td>
<td>01110</td>
<td>13</td>
<td>eieio</td>
</tr>
<tr>
<td>11111</td>
<td>01111</td>
<td>14</td>
<td>mbar</td>
</tr>
<tr>
<td>11111</td>
<td>01100</td>
<td>15</td>
<td>sthepx</td>
</tr>
<tr>
<td>11111</td>
<td>01101</td>
<td>16</td>
<td>eieio</td>
</tr>
<tr>
<td>11111</td>
<td>01110</td>
<td>17</td>
<td>mbar</td>
</tr>
<tr>
<td>11111</td>
<td>01111</td>
<td>18</td>
<td>sthepx</td>
</tr>
</tbody>
</table>
Table 15: Opcode: 31, Extended Opcode: 15

<table>
<thead>
<tr>
<th>Opcode</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>tdp* 64</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>twi*</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>mulli*</td>
<td>B</td>
</tr>
<tr>
<td>01000</td>
<td>sublic*</td>
<td>B</td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td>cmpi*</td>
<td>B</td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>addic*</td>
<td>B</td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td>addic*</td>
<td>B</td>
</tr>
<tr>
<td>01111</td>
<td>addis*</td>
<td>B</td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td>isel</td>
<td></td>
</tr>
</tbody>
</table>
Table 16: (Left) Extended opcodes for primary opcode 59 (instruction bits 21:30)

<table>
<thead>
<tr>
<th></th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</th>
<th>01000</th>
<th>01001</th>
<th>01010</th>
<th>01011</th>
<th>01100</th>
<th>01101</th>
<th>01110</th>
<th>01111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td>dadd</td>
<td>dfmul</td>
<td>dscll</td>
<td>dsci</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>dmpu</td>
<td>dmpu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>dcmpo</td>
<td>dtstex</td>
<td>dscli</td>
<td>dquai</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>dcmpo</td>
<td>dtstex</td>
<td>dscli</td>
<td>dquai</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td>dfdiv</td>
<td>dldiv</td>
<td>dldiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td>dfdiv</td>
<td>dldiv</td>
<td>dldiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1389  Power ISA™ - Book Appendices
Table 16. (Right) Extended opcodes for primary opcode 59 (instruction bits 21:30)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>FP</td>
<td>0001</td>
<td>FP</td>
<td>0010</td>
<td>FP</td>
<td>0011</td>
<td>FP</td>
<td>0100</td>
<td>FP</td>
<td>0101</td>
<td>FP</td>
<td>0110</td>
<td>FP</td>
<td>0111</td>
<td>FP</td>
<td>1000</td>
<td>FP</td>
<td>1001</td>
<td>FP</td>
<td>1010</td>
<td>FP</td>
<td>1011</td>
<td>FP</td>
<td>1010</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `fpys`: FP, A
- `fsubs`: FP, A
- `fadds`: FP, A
- `fsqrts`: FP, A
- `fres`: FP, A
- `fmuls`: FP, A
- `fmsubs`: FP, A
- `fmadda`: FP, A
- `fmsubs`: FP, A
- `fnmadds`: FP, A
### Table 17 (Left) Extended opcodes for primary opcode 60 (instruction bits 21:30)

<table>
<thead>
<tr>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
<td><strong>XX</strong></td>
<td><strong>VX</strong></td>
</tr>
<tr>
<td>0000</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0001</td>
<td>V</td>
<td>S</td>
<td>SUB</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0010</td>
<td>V</td>
<td>S</td>
<td>SUB</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>SUB</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0011</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0100</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0101</td>
<td>V</td>
<td>S</td>
<td>SUB</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>SUB</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0110</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>0111</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1000</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1001</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1010</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1011</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1100</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1101</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1110</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
<tr>
<td>1111</td>
<td>V</td>
<td>S</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
<td>X</td>
<td>MADD</td>
</tr>
</tbody>
</table>
Table 17. (Right) Extended opcodes for primary opcode 60 (instruction bits 21:30)

<table>
<thead>
<tr>
<th>10000</th>
<th>10001</th>
<th>10010</th>
<th>10011</th>
<th>10100</th>
<th>10101</th>
<th>10110</th>
<th>10111</th>
<th>11000</th>
<th>11001</th>
<th>11010</th>
<th>11011</th>
<th>11100</th>
<th>11101</th>
<th>11110</th>
<th>11111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>0000</td>
<td>fcmpu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 18. (Right) Extended opcodes for primary opcode 63 (instruction bits 21:30)

<table>
<thead>
<tr>
<th>00000</th>
<th>10000</th>
<th>10001</th>
<th>10010</th>
<th>10011</th>
<th>10100</th>
<th>10101</th>
<th>10110</th>
<th>10111</th>
<th>11000</th>
<th>11001</th>
<th>11010</th>
<th>11011</th>
<th>11100</th>
<th>11101</th>
<th>11110</th>
<th>11111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |

<table>
<thead>
<tr>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
<th>fp</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv</td>
<td>fsub</td>
<td>fadd</td>
<td>fsgt</td>
<td>fseu</td>
<td>fre</td>
<td>fmul</td>
<td>fragt</td>
<td>fmsub</td>
<td>fmadd</td>
<td>fnmsub</td>
<td>fnmadd</td>
<td>fmsub</td>
<td>fmadd</td>
<td>fnmsub</td>
<td>fnmadd</td>
<td>fmsub</td>
<td>fmadd</td>
<td>fnmsub</td>
<td>fnmadd</td>
</tr>
</tbody>
</table>

1394  Power ISA™ - Book Appendices
Appendix G. Power ISA Instruction Set Sorted by Category

This appendix lists all the instructions in the Power ISA, grouped by category, and in order by mnemonic within category.

<table>
<thead>
<tr>
<th>Format</th>
<th>Opcode</th>
<th>Mode Dep.1</th>
<th>Privilege1</th>
<th>Page</th>
<th>Category1</th>
<th>Mnemonic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>31 7C0001F8</td>
<td></td>
<td></td>
<td>90</td>
<td>64</td>
<td>bpermd Bit Permute Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000074</td>
<td>SR</td>
<td></td>
<td>89</td>
<td>64</td>
<td>cntld[] Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0003D2</td>
<td>SR</td>
<td></td>
<td>76</td>
<td>64</td>
<td>divd[] Divide Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000352</td>
<td>SR</td>
<td></td>
<td>77</td>
<td>64</td>
<td>divide[] Divide Doubleword Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000752</td>
<td>SR</td>
<td></td>
<td>77</td>
<td>64</td>
<td>divideo[] Divide Doubleword Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000312</td>
<td>SR</td>
<td></td>
<td>77</td>
<td>64</td>
<td>divideu[] Divide Doubleword Extended Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000712</td>
<td>SR</td>
<td></td>
<td>77</td>
<td>64</td>
<td>divdeo[] Divide Doubleword Extended Unsigned &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0007D2</td>
<td>SR</td>
<td></td>
<td>76</td>
<td>64</td>
<td>divd[] Divide Doubleword Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000392</td>
<td>SR</td>
<td></td>
<td>76</td>
<td>64</td>
<td>divde[] Divide Doubleword Signed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000792</td>
<td>SR</td>
<td></td>
<td>76</td>
<td>64</td>
<td>divdu[] Divide Doubleword Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007B4</td>
<td>SR</td>
<td></td>
<td>89</td>
<td>64</td>
<td>extsw[] Extend Sign Word</td>
</tr>
<tr>
<td>DS</td>
<td>58 E8000000</td>
<td></td>
<td></td>
<td>53</td>
<td>64</td>
<td>ld Load Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000A8</td>
<td></td>
<td></td>
<td>782</td>
<td>64</td>
<td>ldarx Load Doubleword And Reserve Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000428</td>
<td></td>
<td></td>
<td>60</td>
<td>64</td>
<td>ldbrx Load Doubleword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>DS</td>
<td>58 E8000001</td>
<td></td>
<td></td>
<td>53</td>
<td>64</td>
<td>ldu Load Doubleword with Update</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00006A</td>
<td></td>
<td></td>
<td>53</td>
<td>64</td>
<td>ldux Load Doubleword with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0002A2</td>
<td></td>
<td></td>
<td>53</td>
<td>64</td>
<td>ldx Load Doubleword Indexed</td>
</tr>
<tr>
<td>DS</td>
<td>58 E8000002</td>
<td></td>
<td></td>
<td>52</td>
<td>64</td>
<td>lwa Load Word Algebraic</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0002EA</td>
<td></td>
<td></td>
<td>52</td>
<td>64</td>
<td>lwaux Load Word Algebraic with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0002AA</td>
<td></td>
<td></td>
<td>52</td>
<td>64</td>
<td>lwax Load Word Algebraic Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00092</td>
<td>SR</td>
<td></td>
<td>75</td>
<td>64</td>
<td>mulhd[] Multiply High Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00012</td>
<td>SR</td>
<td></td>
<td>64</td>
<td>64</td>
<td>mulhdu[] Multiply High Doubleword Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D2</td>
<td>SR</td>
<td></td>
<td>64</td>
<td>64</td>
<td>mulhd[] Multiply Low Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0005D2</td>
<td>SR</td>
<td></td>
<td>64</td>
<td>64</td>
<td>mulldo[] Multiply Low Doubleword &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003F4</td>
<td></td>
<td></td>
<td>89</td>
<td>64</td>
<td>popcntd Population Count Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000174</td>
<td></td>
<td></td>
<td>88</td>
<td>64</td>
<td>ptyd Parity Doubleword</td>
</tr>
<tr>
<td>MDS</td>
<td>30 7800010</td>
<td>SR</td>
<td></td>
<td>95</td>
<td>64</td>
<td>ridcl[] Rotate Left Doubleword then Clear Left</td>
</tr>
<tr>
<td>MDS</td>
<td>30 7800012</td>
<td>SR</td>
<td></td>
<td>96</td>
<td>64</td>
<td>ridcr[] Rotate Left Doubleword then Clear Right</td>
</tr>
<tr>
<td>MD</td>
<td>30 7800008</td>
<td>SR</td>
<td></td>
<td>95</td>
<td>64</td>
<td>ridcl[] Rotate Left Doubleword Immediate then Clear</td>
</tr>
<tr>
<td>MD</td>
<td>30 7800000</td>
<td>SR</td>
<td></td>
<td>94</td>
<td>64</td>
<td>ridcl[] Rotate Left Doubleword Immediate then Clear Left</td>
</tr>
<tr>
<td>MD</td>
<td>30 7800004</td>
<td>SR</td>
<td></td>
<td>94</td>
<td>64</td>
<td>ridcr[] Rotate Left Doubleword Immediate then Clear Right</td>
</tr>
<tr>
<td>MD</td>
<td>30 7800000</td>
<td>SR</td>
<td></td>
<td>96</td>
<td>64</td>
<td>ridimi[] Rotate Left Doubleword Immediate then Mask Insert</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000036</td>
<td>SR</td>
<td></td>
<td>99</td>
<td>64</td>
<td>sid[] Shift Left Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000634</td>
<td>SR</td>
<td></td>
<td>100</td>
<td>64</td>
<td>srad[] Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>XS</td>
<td>31 7C000674</td>
<td>SR</td>
<td></td>
<td>100</td>
<td>64</td>
<td>srad[] Shift Right Algebraic Doubleword Immediate</td>
</tr>
<tr>
<td>Format</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>X</td>
<td>7C000436</td>
<td>SR</td>
<td>99</td>
<td>64</td>
<td>srd[]</td>
<td>Shift Right Doubleword</td>
</tr>
<tr>
<td>DS</td>
<td>F8000000</td>
<td>std</td>
<td>57</td>
<td>64</td>
<td>Store Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000528</td>
<td>stdbrx</td>
<td>60</td>
<td>64</td>
<td>Store Doubleword Byte-Reverse Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001AD</td>
<td>stdcx.</td>
<td>782</td>
<td>64</td>
<td>Store Doubleword Conditional Indexed &amp; record CR0</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>F8000001</td>
<td>stdu</td>
<td>57</td>
<td>64</td>
<td>Store Doubleword with Update</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00016A</td>
<td>stdux</td>
<td>57</td>
<td>64</td>
<td>Store Doubleword with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00012A</td>
<td>stdx</td>
<td>57</td>
<td>64</td>
<td>Store Doubleword Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000088</td>
<td>td</td>
<td>81</td>
<td>64</td>
<td>Trap Doubleword</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>08000000</td>
<td>tdi</td>
<td>81</td>
<td>64</td>
<td>Trap Doubleword Immediate</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C000214</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>add[]</td>
<td>Add</td>
</tr>
<tr>
<td>XO</td>
<td>7C000014</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>addc[]</td>
<td>Add Carrying</td>
</tr>
<tr>
<td>XO</td>
<td>7C000414</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>addco[]</td>
<td>Add Carrying &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>7C000114</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>addel[]</td>
<td>Add Extended</td>
</tr>
<tr>
<td>XO</td>
<td>7C000514</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>adddeo[]</td>
<td>Add Extended &amp; record OV &amp; record OV</td>
</tr>
<tr>
<td>D</td>
<td>38000000</td>
<td>add</td>
<td>66</td>
<td>B</td>
<td>Add Immediate</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>30000000</td>
<td>addic</td>
<td>67</td>
<td>B</td>
<td>Add Immediate Carrying</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>34000000</td>
<td>addic.</td>
<td>67</td>
<td>B</td>
<td>Add Immediate Carrying &amp; record CR0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3C000000</td>
<td>addis</td>
<td>66</td>
<td>B</td>
<td>Add Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C0001D4</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>addme[]</td>
<td>Add to Minus One Extended</td>
</tr>
<tr>
<td>XO</td>
<td>7C0005D4</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>addmeo[]</td>
<td>Add to Minus One Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>7C000614</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>addo[]</td>
<td>Add &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>7C000194</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>addze[]</td>
<td>Add to Zero Extended</td>
</tr>
<tr>
<td>XO</td>
<td>7C000594</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>addzeo[]</td>
<td>Add to Zero Extended &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>7C000038</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>and[]</td>
<td>AND</td>
</tr>
<tr>
<td>X</td>
<td>7C000078</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>andc[]</td>
<td>AND with Complement</td>
</tr>
<tr>
<td>D</td>
<td>70000000</td>
<td>SR</td>
<td>82</td>
<td>B</td>
<td>andi.</td>
<td>AND Immediate &amp; record CR0</td>
</tr>
<tr>
<td>D</td>
<td>74000000</td>
<td>SR</td>
<td>82</td>
<td>B</td>
<td>andis.</td>
<td>AND Immediate Shifted &amp; record CR0</td>
</tr>
<tr>
<td>I</td>
<td>48000000</td>
<td>bll[a]</td>
<td>38</td>
<td>B</td>
<td>Branch</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>40000000</td>
<td>bcll[a]</td>
<td>38</td>
<td>B</td>
<td>Branch Conditional</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C003402</td>
<td>bclr[]</td>
<td>39</td>
<td>B</td>
<td>Branch Conditional to Count Register</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C000202</td>
<td>bclr[]</td>
<td>39</td>
<td>B</td>
<td>Branch Conditional to Link Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>4C00460</td>
<td>bctarl[l]</td>
<td>40</td>
<td>B</td>
<td>Branch Conditional to Branch Target Address Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000000</td>
<td>cmp</td>
<td>78</td>
<td>B</td>
<td>Compare</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0003F8</td>
<td>cmpb</td>
<td>86</td>
<td>B</td>
<td>Compare Byte</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2C000000</td>
<td>cmpi</td>
<td>78</td>
<td>B</td>
<td>Compare Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000400</td>
<td>cmpl</td>
<td>79</td>
<td>B</td>
<td>Compare Logical</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2B000000</td>
<td>cmpli</td>
<td>79</td>
<td>B</td>
<td>Compare Logical Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000034</td>
<td>cnlzw[]</td>
<td>85</td>
<td>B</td>
<td>Count Leading Zeros Word</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C002002</td>
<td>crand</td>
<td>41</td>
<td>B</td>
<td>Condition Register AND</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C00102</td>
<td>crandc</td>
<td>42</td>
<td>B</td>
<td>Condition Register AND with Complement</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C00242</td>
<td>creqv</td>
<td>42</td>
<td>B</td>
<td>Condition Register Equivalent</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C001C2</td>
<td>crmand</td>
<td>41</td>
<td>B</td>
<td>Condition Register NAND</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C00042</td>
<td>crnor</td>
<td>42</td>
<td>B</td>
<td>Condition Register NOR</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C00382</td>
<td>cror</td>
<td>41</td>
<td>B</td>
<td>Condition Register OR</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C00342</td>
<td>crorc</td>
<td>42</td>
<td>B</td>
<td>Condition Register OR with Complement</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>4C00182</td>
<td>crxor</td>
<td>41</td>
<td>B</td>
<td>Condition Register XOR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000AC</td>
<td>dcbf</td>
<td>773</td>
<td>B</td>
<td>Data Cache Block Flush</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0006C</td>
<td>dcbst</td>
<td>773</td>
<td>B</td>
<td>Data Cache Block Store</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0022C</td>
<td>dcbt</td>
<td>770</td>
<td>B</td>
<td>Data Cache Block Touch</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C001EC</td>
<td>dcbst</td>
<td>771</td>
<td>B</td>
<td>Data Cache Block Touch for Store</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C007EC</td>
<td>dcbz</td>
<td>773</td>
<td>B</td>
<td>Data Cache Block Zero</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>7C003D6</td>
<td>divw[]</td>
<td>72</td>
<td>B</td>
<td>Divide Word</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C0000356</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>divwe[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C0000756</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>divweo[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C0000316</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>divweu[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C0000716</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>divweuo[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C00007D6</td>
<td>SR</td>
<td>72</td>
<td>B</td>
<td>divwu[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C0000396</td>
<td>SR</td>
<td>72</td>
<td>B</td>
<td>divwuo[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0000796</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>extsb[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0000714</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>extsh[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00007AC</td>
<td></td>
<td>762</td>
<td>B</td>
<td>icbi</td>
</tr>
<tr>
<td>A</td>
<td>31</td>
<td>7C00001E</td>
<td>B</td>
<td>81</td>
<td>B</td>
<td>isel</td>
</tr>
<tr>
<td>X</td>
<td>19</td>
<td>4C000012C</td>
<td>B</td>
<td>776</td>
<td>B</td>
<td>lsync</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000068</td>
<td>B</td>
<td>777</td>
<td>B</td>
<td>lbarx</td>
</tr>
<tr>
<td>D</td>
<td>34</td>
<td>880000000</td>
<td>B</td>
<td>48</td>
<td>B</td>
<td>lbz</td>
</tr>
<tr>
<td>D</td>
<td>35</td>
<td>8C0000000</td>
<td>B</td>
<td>48</td>
<td>B</td>
<td>lbzu</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00000EE</td>
<td>B</td>
<td>48</td>
<td>B</td>
<td>lbzux</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00000AE</td>
<td>B</td>
<td>49</td>
<td>B</td>
<td>lbzx</td>
</tr>
<tr>
<td>D</td>
<td>42</td>
<td>A80000000</td>
<td>B</td>
<td>50</td>
<td>B</td>
<td>lha</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0000E8</td>
<td>B</td>
<td>778</td>
<td>B</td>
<td>lharx</td>
</tr>
<tr>
<td>D</td>
<td>43</td>
<td>A00000000</td>
<td>B</td>
<td>50</td>
<td>B</td>
<td>lhau</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002EE</td>
<td>B</td>
<td>50</td>
<td>B</td>
<td>lhaux</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002AE</td>
<td>B</td>
<td>50</td>
<td>B</td>
<td>lhax</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000062C</td>
<td>B</td>
<td>59</td>
<td>B</td>
<td>lhbrx</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>A00000000</td>
<td>B</td>
<td>49</td>
<td>B</td>
<td>lhiz</td>
</tr>
<tr>
<td>D</td>
<td>41</td>
<td>A40000000</td>
<td>B</td>
<td>49</td>
<td>B</td>
<td>lhizu</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000026E</td>
<td>B</td>
<td>49</td>
<td>B</td>
<td>lhzux</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000022E</td>
<td>B</td>
<td>49</td>
<td>B</td>
<td>lhzx</td>
</tr>
<tr>
<td>D</td>
<td>46</td>
<td>B80000000</td>
<td>B</td>
<td>61</td>
<td>B</td>
<td>lmw</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000028</td>
<td>B</td>
<td>777</td>
<td>B</td>
<td>lwbarx</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000042C</td>
<td>B</td>
<td>59</td>
<td>B</td>
<td>lwbrx</td>
</tr>
<tr>
<td>D</td>
<td>32</td>
<td>800000000</td>
<td>B</td>
<td>51</td>
<td>B</td>
<td>lwz</td>
</tr>
<tr>
<td>D</td>
<td>33</td>
<td>840000000</td>
<td>B</td>
<td>51</td>
<td>B</td>
<td>lwzu</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000006E</td>
<td>B</td>
<td>51</td>
<td>B</td>
<td>lwzux</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002BE</td>
<td>B</td>
<td>51</td>
<td>B</td>
<td>lwzx</td>
</tr>
<tr>
<td>X</td>
<td>19</td>
<td>4C0000000</td>
<td>B</td>
<td>42</td>
<td>B</td>
<td>mcrf</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000026E</td>
<td>B</td>
<td>108</td>
<td>B</td>
<td>mfcr</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C1000026</td>
<td>B</td>
<td>112</td>
<td>B</td>
<td>mfocrf</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0002A6</td>
<td>O</td>
<td>106</td>
<td>B</td>
<td>mfspr</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000120</td>
<td>B</td>
<td>107</td>
<td>B</td>
<td>mtcrf</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C100120</td>
<td>B</td>
<td>112</td>
<td>B</td>
<td>mtocrf</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0003A6</td>
<td>O</td>
<td>104</td>
<td>B</td>
<td>mtspcr</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000096</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td>mulhw[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000016</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td>mulhwu[]</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>1C0000000</td>
<td>B</td>
<td>71</td>
<td>B</td>
<td>multi</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0001D6</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td>mulw[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0005D6</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td>mulwo[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0003B8</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>nand[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0000DD</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>neg[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0004D0</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>nego[]</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>X 31</td>
<td>7C0000F8</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>nor[,]</td>
<td>NOR</td>
</tr>
<tr>
<td>X 31</td>
<td>7C000378</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>or[,]</td>
<td>OR</td>
</tr>
<tr>
<td>X 31</td>
<td>7C000338</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>orc[,]</td>
<td>OR with Complement</td>
</tr>
<tr>
<td>D 24</td>
<td>60000000</td>
<td>82</td>
<td>B</td>
<td>on</td>
<td>OR Immediate</td>
<td></td>
</tr>
<tr>
<td>D 25</td>
<td>64000000</td>
<td>83</td>
<td>B</td>
<td>ons</td>
<td>OR Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0000F4</td>
<td>SR</td>
<td>87</td>
<td>B</td>
<td>popcntb</td>
<td>Population Count Byte-wise</td>
</tr>
<tr>
<td>X 31</td>
<td>7C0002F4</td>
<td>SR</td>
<td>87</td>
<td>B</td>
<td>popcntw</td>
<td>Population Count Words</td>
</tr>
<tr>
<td>X 31</td>
<td>7C0000B8</td>
<td>85</td>
<td>B</td>
<td>prtyw</td>
<td>Parity Word</td>
<td></td>
</tr>
<tr>
<td>M 20</td>
<td>50000000</td>
<td>SR</td>
<td>93</td>
<td>B</td>
<td>rlwinm[,]</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
</tr>
<tr>
<td>M 21</td>
<td>54000000</td>
<td>SR</td>
<td>91</td>
<td>B</td>
<td>rlwinm[,]</td>
<td>Rotate Left Word Immediate then AND with Mask</td>
</tr>
<tr>
<td>M 23</td>
<td>5C000000</td>
<td>SR</td>
<td>92</td>
<td>B</td>
<td>rlwinm[,]</td>
<td>Rotate Left Word then AND with Mask</td>
</tr>
<tr>
<td>SC 17</td>
<td>44000000</td>
<td>B</td>
<td>43</td>
<td>463 1040</td>
<td>sc</td>
<td>System Call</td>
</tr>
<tr>
<td>X 31</td>
<td>7C000030</td>
<td>SR</td>
<td>97</td>
<td>B</td>
<td>slw[,]</td>
<td>Shift Left Word</td>
</tr>
<tr>
<td>X 31</td>
<td>7C000630</td>
<td>SR</td>
<td>96</td>
<td>B</td>
<td>sraw[,]</td>
<td>Shift Right Algebraic Word</td>
</tr>
<tr>
<td>X 31</td>
<td>7C000670</td>
<td>SR</td>
<td>96</td>
<td>B</td>
<td>sraw[,]</td>
<td>Shift Right Algebraic Word Immediate</td>
</tr>
<tr>
<td>X 31</td>
<td>7C000430</td>
<td>SR</td>
<td>97</td>
<td>B</td>
<td>srw[,]</td>
<td>Shift Right Word</td>
</tr>
<tr>
<td>D 38</td>
<td>98000000</td>
<td>54</td>
<td>B</td>
<td>sib</td>
<td>Store Byte</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00056D</td>
<td>779</td>
<td>B</td>
<td>sbcx[,]</td>
<td>Store Byte Conditional Indexed</td>
<td></td>
</tr>
<tr>
<td>D 39</td>
<td>9C000000</td>
<td>54</td>
<td>B</td>
<td>sbu</td>
<td>Store Byte with Update</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0001EE</td>
<td>54</td>
<td>B</td>
<td>sbux</td>
<td>Store Byte with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0001AE</td>
<td>54</td>
<td>B</td>
<td>sbx</td>
<td>Store Byte Indexed</td>
<td></td>
</tr>
<tr>
<td>D 44</td>
<td>B0000000</td>
<td>55</td>
<td>B</td>
<td>sht</td>
<td>Store Halfword</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00072C</td>
<td>59</td>
<td>B</td>
<td>sbhbrx</td>
<td>Store Halfword Byte-Reverse Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0005AD</td>
<td>780</td>
<td>B</td>
<td>stbcx[,]</td>
<td>Store Halfword Conditional Indexed Xform</td>
<td></td>
</tr>
<tr>
<td>D 45</td>
<td>B4000000</td>
<td>55</td>
<td>B</td>
<td>sthu</td>
<td>Store Halfword with Update</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00036E</td>
<td>55</td>
<td>B</td>
<td>sthux</td>
<td>Store Halfword with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00032E</td>
<td>55</td>
<td>B</td>
<td>sthx</td>
<td>Store Halfword Indexed</td>
<td></td>
</tr>
<tr>
<td>D 47</td>
<td>BC000000</td>
<td>61</td>
<td>B</td>
<td>stmw</td>
<td>Store Multiple Word</td>
<td></td>
</tr>
<tr>
<td>D 36</td>
<td>90000000</td>
<td>56</td>
<td>B</td>
<td>sty</td>
<td>Store Word</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00052C</td>
<td>59</td>
<td>B</td>
<td>stwbrx</td>
<td>Store Word Byte-Reverse Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00012D</td>
<td>781</td>
<td>B</td>
<td>stwcx[,]</td>
<td>Store Word Conditional Indexed &amp; record CR0</td>
<td></td>
</tr>
<tr>
<td>D 37</td>
<td>94000000</td>
<td>56</td>
<td>B</td>
<td>stwu</td>
<td>Store Word with Update</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00016E</td>
<td>56</td>
<td>B</td>
<td>stwux</td>
<td>Store Word with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00012E</td>
<td>56</td>
<td>B</td>
<td>stwx</td>
<td>Store Word Indexed</td>
<td></td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000050</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>sub[,]</td>
<td>Subtract From</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000010</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>subfc[,]</td>
<td>Subtract From Carrying</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000410</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>subfc[,]</td>
<td>Subtract From Carrying &amp; record OV</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000110</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfe[,]</td>
<td>Subtract From Extended</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000510</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfe[,]</td>
<td>Subtract From Extended &amp; record OV</td>
</tr>
<tr>
<td>D 8</td>
<td>20000000</td>
<td>68</td>
<td>B</td>
<td>subfic</td>
<td>Subtract From Immediate Carrying</td>
<td></td>
</tr>
<tr>
<td>XO 31</td>
<td>7C0001D0</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfme[,]</td>
<td>Subtract From Minus One Extended</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C0005D0</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfme[,]</td>
<td>Subtract From Minus One Extended &amp; record OV</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000450</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>subfo[,]</td>
<td>Subtract From &amp; record OV</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000190</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>subfze[,]</td>
<td>Subtract From Zero Extended</td>
</tr>
<tr>
<td>XO 31</td>
<td>7C000590</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>subfze[,]</td>
<td>Subtract From Zero Extended &amp; record OV</td>
</tr>
<tr>
<td>X 31</td>
<td>7C0004AC</td>
<td>786</td>
<td>B</td>
<td>sync</td>
<td>Synchronize</td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C000080</td>
<td>80</td>
<td>B</td>
<td>tw</td>
<td>Trap Word</td>
<td></td>
</tr>
<tr>
<td>D 3</td>
<td>00000000</td>
<td>80</td>
<td>B</td>
<td>twi</td>
<td>Trap Word Immediate</td>
<td></td>
</tr>
<tr>
<td>X 26</td>
<td>68000000</td>
<td>B</td>
<td>xnop</td>
<td>Executed No Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C000278</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>xor[,]</td>
<td>XOR</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>D</td>
<td>26</td>
<td>68000000</td>
<td>83</td>
<td>B</td>
<td>xori</td>
<td>XOR Immediate</td>
</tr>
<tr>
<td>D</td>
<td>27</td>
<td>6C000000</td>
<td>83</td>
<td>B</td>
<td>xoris</td>
<td>XOR Immediate Shifted</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C000094</td>
<td>H 101</td>
<td>BCD</td>
<td>addg6s</td>
<td>Add and Generate Sixes</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000274</td>
<td>H 101</td>
<td>BCD</td>
<td>cbcdtd</td>
<td>Convert Binary Coded Decimal To Declets</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000234</td>
<td>H 101</td>
<td>BCD</td>
<td>cdtdbcd</td>
<td>Convert Declets To Binary Coded Decimal</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000004</td>
<td>289</td>
<td>DFP</td>
<td>dadd[.]</td>
<td>Decimal Floating Add</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000004</td>
<td>289</td>
<td>DFP</td>
<td>daddq[.]</td>
<td>Decimal Floating Add Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000644</td>
<td>311</td>
<td>DFP</td>
<td>dcfIix[.]</td>
<td>Decimal Floating Convert From Fixed</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000644</td>
<td>311</td>
<td>DFP</td>
<td>dcfIxq[.]</td>
<td>Decimal Floating Convert From Fixed Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000104</td>
<td>295</td>
<td>DFP</td>
<td>dcmo</td>
<td>Decimal Floating Compare Ordered</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000104</td>
<td>295</td>
<td>DFP</td>
<td>dcmPq</td>
<td>Decimal Floating Compare Ordered Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000504</td>
<td>294</td>
<td>DFP</td>
<td>dcmPu</td>
<td>Decimal Floating Compare Unordered</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000504</td>
<td>295</td>
<td>DFP</td>
<td>dcmPuq</td>
<td>Decimal Floating Compare Unordered Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000204</td>
<td>309</td>
<td>DFP</td>
<td>dctlp[.]</td>
<td>Decimal Floating Convert To DFP Long</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000244</td>
<td>311</td>
<td>DFP</td>
<td>dcfix[.]</td>
<td>Decimal Floating Convert To Fixed</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000244</td>
<td>311</td>
<td>DFP</td>
<td>dcfIxq[.]</td>
<td>Decimal Floating Convert To Fixed Quad</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000204</td>
<td>309</td>
<td>DFP</td>
<td>dctlpq[.]</td>
<td>Decimal Floating Convert To DFP Extended</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000284</td>
<td>313</td>
<td>DFP</td>
<td>ddedpd[.]</td>
<td>Decimal Floating Decode DPD To BCD</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000284</td>
<td>313</td>
<td>DFP</td>
<td>ddedpdq[.]</td>
<td>Decimal Floating Decode DPD To BCD Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000444</td>
<td>292</td>
<td>DFP</td>
<td>ddIiv[.]</td>
<td>Decimal Floating Divide</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000444</td>
<td>292</td>
<td>DFP</td>
<td>ddIivq[.]</td>
<td>Decimal Floating Divide Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000684</td>
<td>313</td>
<td>DFP</td>
<td>dencbcd[.]</td>
<td>Decimal Floating Encode BCD To DPD</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000684</td>
<td>313</td>
<td>DFP</td>
<td>dencbcdq[.]</td>
<td>Decimal Floating Encode BCD To DPD Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC0006C4</td>
<td>314</td>
<td>DFP</td>
<td>diex[.]</td>
<td>Decimal Floating Insert Exponent</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC0006C4</td>
<td>314</td>
<td>DFP</td>
<td>diexq[.]</td>
<td>Decimal Floating Insert Exponent Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000044</td>
<td>291</td>
<td>DFP</td>
<td>dmul[.]</td>
<td>Decimal Floating Multiply</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000044</td>
<td>291</td>
<td>DFP</td>
<td>dmulq[.]</td>
<td>Decimal Floating Multiply Quad</td>
</tr>
<tr>
<td>Z23</td>
<td>59</td>
<td>EC000006</td>
<td>300</td>
<td>DFP</td>
<td>dqua[.]</td>
<td>Decimal Quantize</td>
</tr>
<tr>
<td>Z23</td>
<td>59</td>
<td>EC000086</td>
<td>299</td>
<td>DFP</td>
<td>dquaI[.]</td>
<td>Decimal Quantize Immediate</td>
</tr>
<tr>
<td>Z23</td>
<td>63</td>
<td>FC000086</td>
<td>299</td>
<td>DFP</td>
<td>dquaIq[.]</td>
<td>Decimal Quantize Immediate Quad</td>
</tr>
<tr>
<td>Z23</td>
<td>63</td>
<td>FC000006</td>
<td>300</td>
<td>DFP</td>
<td>dquaq[.]</td>
<td>Decimal Quantize Quad</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000604</td>
<td>310</td>
<td>DFP</td>
<td>drdpq[.]</td>
<td>Decimal Floating Round To DFP Long</td>
</tr>
<tr>
<td>Z23</td>
<td>59</td>
<td>EC0001C6</td>
<td>307</td>
<td>DFP</td>
<td>drIn[.]</td>
<td>Decimal Floating Round To FP Integer Without Inexact</td>
</tr>
<tr>
<td>Z23</td>
<td>63</td>
<td>FC0001C6</td>
<td>307</td>
<td>DFP</td>
<td>drInq[.]</td>
<td>Decimal Floating Round To FP Integer Without Inexact Quad</td>
</tr>
<tr>
<td>Z23</td>
<td>59</td>
<td>EC0000C6</td>
<td>305</td>
<td>DFP</td>
<td>drInx[.]</td>
<td>Decimal Floating Round To FP Integer With Inexact</td>
</tr>
<tr>
<td>Z23</td>
<td>63</td>
<td>FC0000C6</td>
<td>305</td>
<td>DFP</td>
<td>drInxq[.]</td>
<td>Decimal Floating Round To FP Integer With Inexact Quad</td>
</tr>
<tr>
<td>Z23</td>
<td>59</td>
<td>EC000046</td>
<td>302</td>
<td>DFP</td>
<td>drrmd[.]</td>
<td>Decimal Floating Reround</td>
</tr>
<tr>
<td>Z23</td>
<td>63</td>
<td>FC000046</td>
<td>302</td>
<td>DFP</td>
<td>drrmdq[.]</td>
<td>Decimal Floating Reround Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000604</td>
<td>310</td>
<td>DFP</td>
<td>drsp[.]</td>
<td>Decimal Floating Round To DFP Short</td>
</tr>
<tr>
<td>Z22</td>
<td>59</td>
<td>EC000084</td>
<td>316</td>
<td>DFP</td>
<td>dsclI[.]</td>
<td>Decimal Floating Shift Coefficient Left Immediate</td>
</tr>
<tr>
<td>Z22</td>
<td>63</td>
<td>FC000084</td>
<td>316</td>
<td>DFP</td>
<td>dsclIq[.]</td>
<td>Decimal Floating Shift Coefficient Left Immediate Quad</td>
</tr>
<tr>
<td>Z22</td>
<td>59</td>
<td>EC0000C4</td>
<td>316</td>
<td>DFP</td>
<td>dscri[.]</td>
<td>Decimal Floating Shift Coefficient Right Immediate</td>
</tr>
<tr>
<td>Z22</td>
<td>63</td>
<td>FC0000C4</td>
<td>316</td>
<td>DFP</td>
<td>dscriq[.]</td>
<td>Decimal Floating Shift Coefficient Right Immediate Quad</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000404</td>
<td>289</td>
<td>DFP</td>
<td>dsub[.]</td>
<td>Decimal Floating Subtract</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000404</td>
<td>289</td>
<td>DFP</td>
<td>dsubbq[.]</td>
<td>Decimal Floating Subtract Quad</td>
</tr>
<tr>
<td>Z22</td>
<td>59</td>
<td>EC000184</td>
<td>296</td>
<td>DFP</td>
<td>dtstdc</td>
<td>Decimal Floating Test Data Class</td>
</tr>
<tr>
<td>Z22</td>
<td>63</td>
<td>FC000184</td>
<td>296</td>
<td>DFP</td>
<td>dtstdcq</td>
<td>Decimal Floating Test Data Class Quad</td>
</tr>
<tr>
<td>Format</td>
<td>Primary</td>
<td>Base (hex)</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>Z22</td>
<td>59</td>
<td>EC0001C4</td>
<td>DFP</td>
<td>dtstdg</td>
<td>296</td>
<td>DFP</td>
</tr>
<tr>
<td>Z22</td>
<td>63</td>
<td>FC0001C4</td>
<td>DFP</td>
<td>dtstdgq</td>
<td>296</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000144</td>
<td>DFP</td>
<td>dtstex</td>
<td>297</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000144</td>
<td>DFP</td>
<td>dtstexq</td>
<td>297</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC000544</td>
<td>DFP</td>
<td>dtstsf</td>
<td>298</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC000544</td>
<td>DFP</td>
<td>dtstsfq</td>
<td>298</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC0002C4</td>
<td>DFP</td>
<td>dsex[.]</td>
<td>314</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>63</td>
<td>FC0002C4</td>
<td>DFP</td>
<td>dsexq[.]</td>
<td>314</td>
<td>DFP</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0003C6</td>
<td>DS</td>
<td>dsn</td>
<td>824</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000406</td>
<td>DS</td>
<td>lbdx</td>
<td>822</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00046C</td>
<td>DS</td>
<td>lddx</td>
<td>822</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000646</td>
<td>DS</td>
<td>ifddx</td>
<td>822</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000446</td>
<td>DS</td>
<td>lhdx</td>
<td>822</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000486</td>
<td>DS</td>
<td>lwdx</td>
<td>822</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000506</td>
<td>DS</td>
<td>stbdbx</td>
<td>823</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00056C</td>
<td>DS</td>
<td>stddx</td>
<td>823</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000746</td>
<td>DS</td>
<td>stffdx</td>
<td>823</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000546</td>
<td>DS</td>
<td>sthtdx</td>
<td>823</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000586</td>
<td>DS</td>
<td>stwdx</td>
<td>823</td>
<td>DS</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0005EC</td>
<td>E</td>
<td>dcba</td>
<td>770</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0003AC</td>
<td>P</td>
<td>dbci</td>
<td>1118</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002C</td>
<td>E</td>
<td>dnh</td>
<td>1228</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00022C</td>
<td>E</td>
<td>icbt</td>
<td>762</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0006AC</td>
<td>E</td>
<td>mbar</td>
<td>790</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000400</td>
<td>E</td>
<td>mrdr</td>
<td>113</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000066</td>
<td>P</td>
<td>rfc</td>
<td>1041</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000064</td>
<td>P</td>
<td>rfi</td>
<td>1041</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00004C</td>
<td>P</td>
<td>rfmci</td>
<td>1042</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000024</td>
<td>P</td>
<td>tblbx</td>
<td>1134</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000624</td>
<td>P</td>
<td>tblivax</td>
<td>1132</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000764</td>
<td>P</td>
<td>tblre</td>
<td>1139</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000724</td>
<td>P</td>
<td>tblsx</td>
<td>1136</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0001A4</td>
<td>P</td>
<td>tblwe</td>
<td>1141</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000106</td>
<td>E</td>
<td>wrtee</td>
<td>1056</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000146</td>
<td>E</td>
<td>wrteel</td>
<td>1057</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00028C</td>
<td>P</td>
<td>dcread</td>
<td>1242</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0003CC</td>
<td>P</td>
<td>dcread</td>
<td>1242</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0007CC</td>
<td>P</td>
<td>icread</td>
<td>1243</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00038C</td>
<td>P</td>
<td>dci</td>
<td>1239</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00078C</td>
<td>P</td>
<td>ici</td>
<td>1239</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000286</td>
<td>P</td>
<td>mfdcr</td>
<td>1055</td>
<td>E.DC</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000246</td>
<td>E.DC</td>
<td>mfdcrux</td>
<td>113</td>
<td>E.DC</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000206</td>
<td>E.DC</td>
<td>mfdcrx</td>
<td>1055</td>
<td>E.DC</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000386</td>
<td>E.DC</td>
<td>mtddcr</td>
<td>1054</td>
<td>E.DC</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000346</td>
<td>E.DC</td>
<td>mtddcrx</td>
<td>113</td>
<td>E.DC</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000306</td>
<td>E.DC</td>
<td>mtddcrx</td>
<td>1054</td>
<td>E.DC</td>
</tr>
<tr>
<td>X</td>
<td>19</td>
<td>4C00004E</td>
<td>E.DC</td>
<td>rdfi</td>
<td>1042</td>
<td>E</td>
</tr>
<tr>
<td>XL</td>
<td>31</td>
<td>7C00021C</td>
<td>E.HV</td>
<td>ehpriv</td>
<td>1043</td>
<td>E.HV</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C00000C</td>
<td>E.HV</td>
<td>rfgi</td>
<td>1043</td>
<td>E.HV</td>
</tr>
</tbody>
</table>

1400  Power ISA™ - Book Appendices
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mode Dep.1</th>
<th>Privilege1</th>
<th>Page</th>
<th>Category1</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>X 31 7C00007E</td>
<td>P</td>
<td>1063</td>
<td>E.PD</td>
<td>dcbstep</td>
<td>Data Cache Block Store by External PID</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00027E</td>
<td>P</td>
<td>1063</td>
<td>E.PD</td>
<td>dcbtep</td>
<td>Data Cache Block Touch by External PID</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0001FE</td>
<td>P</td>
<td>1066</td>
<td>E.PD</td>
<td>dcbsttep</td>
<td>Data Cache Block Touch for Store by External PID</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0007FE</td>
<td>P</td>
<td>1067</td>
<td>E.PD</td>
<td>dcbsp</td>
<td>Data Cache Block Zero by External PID</td>
<td></td>
</tr>
<tr>
<td>EVX 31 7C00063E</td>
<td>P</td>
<td>1069</td>
<td>E.PD</td>
<td>evlddepex</td>
<td>Vector Load Double Word into Double Word by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX 31 7C00073E</td>
<td>P</td>
<td>1069</td>
<td>E.PD</td>
<td>evstddepex</td>
<td>Vector Store Double of Double by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0007BE</td>
<td>P</td>
<td>1067</td>
<td>E.PD</td>
<td>icbiep</td>
<td>Instruction Cache Block Invalidate by External PID</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0000BE</td>
<td>P</td>
<td>1059</td>
<td>E.PD</td>
<td>lbepx</td>
<td>Load Byte and Zero by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0004BE</td>
<td>P</td>
<td>1068</td>
<td>E.PD</td>
<td>lfdepex</td>
<td>Load Floating-Point Double by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00023E</td>
<td>P</td>
<td>1059</td>
<td>E.PD</td>
<td>lhepx</td>
<td>Load Halfword and Zero by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00024E</td>
<td>P</td>
<td>1070</td>
<td>E.PD</td>
<td>lvexpx</td>
<td>Load Vector by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00020E</td>
<td>P</td>
<td>1060</td>
<td>E.PD</td>
<td>lwepx</td>
<td>Load Word and Zero by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0001BE</td>
<td>P</td>
<td>1061</td>
<td>E.PD</td>
<td>stbepx</td>
<td>Store Byte by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0005BE</td>
<td>P</td>
<td>1068</td>
<td>E.PD</td>
<td>stfdepex</td>
<td>Store Floating-Point Double by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00033E</td>
<td>P</td>
<td>1061</td>
<td>E.PD</td>
<td>sthepx</td>
<td>Store Halfword by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00064E</td>
<td>P</td>
<td>1071</td>
<td>E.PD</td>
<td>stvepex</td>
<td>Store Vector by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00060E</td>
<td>P</td>
<td>1071</td>
<td>E.PD</td>
<td>stvepxl</td>
<td>Store Vector by External PID Indexed Last</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00013E</td>
<td>P</td>
<td>1062</td>
<td>E.PD</td>
<td>lwepx</td>
<td>Load Word and Zero by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0003A</td>
<td>P</td>
<td>1060</td>
<td>E.PD:64</td>
<td>ldepex</td>
<td>Load Doubleword by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00013A</td>
<td>P</td>
<td>1062</td>
<td>E.PD:64</td>
<td>stdepex</td>
<td>Store Doubleword by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>XFX 31 7C00029C</td>
<td>O</td>
<td>1257</td>
<td>E.PM</td>
<td>mfpmr</td>
<td>Move from Performance Monitor Register</td>
<td></td>
</tr>
<tr>
<td>XFX 31 7C00039C</td>
<td>O</td>
<td>1257</td>
<td>E.PM</td>
<td>mtmpr</td>
<td>Move To Performance Monitor Register</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0006A5</td>
<td>P</td>
<td>1138</td>
<td>E.TWC</td>
<td>tbsrx</td>
<td>TLB Search and Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00026C</td>
<td>EC</td>
<td>526</td>
<td>ECIWX</td>
<td>ecixw</td>
<td>External Control In Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00036C</td>
<td>EC</td>
<td>526</td>
<td>ECIWX</td>
<td>ecowx</td>
<td>External Control Out Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00030C</td>
<td>M</td>
<td>1123</td>
<td>ECL</td>
<td>dcblic</td>
<td>Data Cache Block Lock Clear</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00034D</td>
<td>M</td>
<td>1121</td>
<td>ECL</td>
<td>dcbblkq</td>
<td>Data Cache Block Lock Query</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00010C</td>
<td>M</td>
<td>1122</td>
<td>ECL</td>
<td>dcbtlss</td>
<td>Data Cache Block Touch and Lock Set</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0001CC</td>
<td>M</td>
<td>1124</td>
<td>ECL</td>
<td>icbiclic</td>
<td>Instruction Cache Block Lock Clear</td>
<td></td>
</tr>
<tr>
<td>X 31 7C00018D</td>
<td>M</td>
<td>1121</td>
<td>ECL</td>
<td>icblqk</td>
<td>Instruction Cache Block Lock Query</td>
<td></td>
</tr>
<tr>
<td>X 31 7C0003CC</td>
<td>M</td>
<td>1123</td>
<td>ECL</td>
<td>icbtls</td>
<td>Instruction Cache Block Touch and Lock Set</td>
<td></td>
</tr>
<tr>
<td>X 63 FC000040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Floating Compare Ordered</td>
</tr>
<tr>
<td>X 63 FC000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Floating Compare Unordered</td>
</tr>
<tr>
<td>X 63 FC000100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Floating Test for software Divide</td>
</tr>
<tr>
<td>X 63 FC000140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Floating Test for software Square Root</td>
</tr>
<tr>
<td>D 50 C8000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double</td>
</tr>
<tr>
<td>D 51 CC000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double with Update</td>
</tr>
<tr>
<td>X 31 7C0004EE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double with Update Indexed</td>
</tr>
<tr>
<td>X 31 7C0004AE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Double Indexed</td>
</tr>
<tr>
<td>X 31 7C0006AE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point as Integer Word Algebraic Indexed</td>
</tr>
<tr>
<td>X 31 7C00066E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point as Integer Word and Zero Indexed</td>
</tr>
<tr>
<td>D 48 C0000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Single</td>
</tr>
<tr>
<td>D 49 CA000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Load Floating-Point Single with Update</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>31 TC00046E</td>
<td></td>
<td></td>
<td>138</td>
<td>FP</td>
<td>lfsux</td>
</tr>
<tr>
<td>X</td>
<td>31 TC00042E</td>
<td></td>
<td></td>
<td>138</td>
<td>FP</td>
<td>lfx</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000080</td>
<td></td>
<td></td>
<td>161</td>
<td>FP</td>
<td>mcrfs</td>
</tr>
<tr>
<td>D</td>
<td>54 D8000000</td>
<td></td>
<td></td>
<td>139</td>
<td>FP</td>
<td>stdf</td>
</tr>
<tr>
<td>D</td>
<td>55 DC000000</td>
<td></td>
<td></td>
<td>139</td>
<td>FP</td>
<td>stdfu</td>
</tr>
<tr>
<td>X</td>
<td>31 TC0005EE</td>
<td></td>
<td></td>
<td>139</td>
<td>FP</td>
<td>stdfux</td>
</tr>
<tr>
<td>X</td>
<td>31 TC0005AE</td>
<td></td>
<td></td>
<td>139</td>
<td>FP</td>
<td>stdfx</td>
</tr>
<tr>
<td>X</td>
<td>31 TC0007AE</td>
<td></td>
<td></td>
<td>140</td>
<td>FP</td>
<td>stdiwx</td>
</tr>
<tr>
<td>D</td>
<td>52 D0000000</td>
<td></td>
<td></td>
<td>138</td>
<td>FP</td>
<td>stds</td>
</tr>
<tr>
<td>D</td>
<td>53 D4000000</td>
<td></td>
<td></td>
<td>138</td>
<td>FP</td>
<td>stdsu</td>
</tr>
<tr>
<td>X</td>
<td>31 TC00056E</td>
<td></td>
<td></td>
<td>138</td>
<td>FP</td>
<td>stdsux</td>
</tr>
<tr>
<td>DS</td>
<td>57 E4000000</td>
<td></td>
<td></td>
<td>141</td>
<td>FP.out</td>
<td>lfdp</td>
</tr>
<tr>
<td>X</td>
<td>31 TC00062E</td>
<td></td>
<td></td>
<td>141</td>
<td>FP.out</td>
<td>lfdpx</td>
</tr>
<tr>
<td>DS</td>
<td>61 F4000000</td>
<td></td>
<td></td>
<td>141</td>
<td>FP</td>
<td>stdfp</td>
</tr>
<tr>
<td>X</td>
<td>31 TC00072E</td>
<td></td>
<td></td>
<td>141</td>
<td>FP</td>
<td>stdfpx</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000210</td>
<td></td>
<td></td>
<td>142</td>
<td>FP[R]</td>
<td>fabs[]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC00002A</td>
<td></td>
<td></td>
<td>144</td>
<td>FP[R]</td>
<td>fadd[]</td>
</tr>
<tr>
<td>A</td>
<td>59 EC00002A</td>
<td></td>
<td></td>
<td>144</td>
<td>FP[R]</td>
<td>fadd[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00069C</td>
<td></td>
<td></td>
<td>155</td>
<td>FP[R]</td>
<td>fcfs[]</td>
</tr>
<tr>
<td>X</td>
<td>59 EC00069C</td>
<td></td>
<td></td>
<td>156</td>
<td>FP[R]</td>
<td>fcfs[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00079C</td>
<td></td>
<td></td>
<td>156</td>
<td>FP[R]</td>
<td>fcfsu[]</td>
</tr>
<tr>
<td>X</td>
<td>59 EC00079C</td>
<td></td>
<td></td>
<td>157</td>
<td>FP[R]</td>
<td>fcfsus[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000101</td>
<td></td>
<td></td>
<td>142</td>
<td>FP[R]</td>
<td>fcpsgn[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00065C</td>
<td></td>
<td></td>
<td>151</td>
<td>FP[R]</td>
<td>fctid[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00075C</td>
<td></td>
<td></td>
<td>152</td>
<td>FP[R]</td>
<td>fctid[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00075E</td>
<td></td>
<td></td>
<td>153</td>
<td>FP[R]</td>
<td>fctiduz[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00065E</td>
<td></td>
<td></td>
<td>152</td>
<td>FP[R]</td>
<td>fctidz[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00011C</td>
<td></td>
<td></td>
<td>153</td>
<td>FP[R]</td>
<td>fctiw[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00011E</td>
<td></td>
<td></td>
<td>154</td>
<td>FP[R]</td>
<td>fctiw[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00001E</td>
<td></td>
<td></td>
<td>155</td>
<td>FP[R]</td>
<td>fctiwz[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00001E</td>
<td></td>
<td></td>
<td>154</td>
<td>FP[R]</td>
<td>fctiwz[]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC000024</td>
<td></td>
<td></td>
<td>145</td>
<td>FP[R]</td>
<td>fdiv[]</td>
</tr>
<tr>
<td>A</td>
<td>59 EC000024</td>
<td></td>
<td></td>
<td>145</td>
<td>FP[R]</td>
<td>fdiv[]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC00003A</td>
<td></td>
<td></td>
<td>149</td>
<td>FP[R]</td>
<td>fmadd[]</td>
</tr>
<tr>
<td>A</td>
<td>59 EC00003A</td>
<td></td>
<td></td>
<td>149</td>
<td>FP[R]</td>
<td>fmadds[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000090</td>
<td></td>
<td></td>
<td>142</td>
<td>FP[R]</td>
<td>fmrr[]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC000038</td>
<td></td>
<td></td>
<td>149</td>
<td>FP[R]</td>
<td>fmsub[]</td>
</tr>
<tr>
<td>A</td>
<td>59 EC000038</td>
<td></td>
<td></td>
<td>149</td>
<td>FP[R]</td>
<td>fmsubs[]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC000032</td>
<td></td>
<td></td>
<td>145</td>
<td>FP[R]</td>
<td>fm][[]</td>
</tr>
<tr>
<td>A</td>
<td>59 EC000032</td>
<td></td>
<td></td>
<td>145</td>
<td>FP[R]</td>
<td>fmuls[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000110</td>
<td></td>
<td></td>
<td>142</td>
<td>FP[R]</td>
<td>fnabs[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000050</td>
<td></td>
<td></td>
<td>142</td>
<td>FP[R]</td>
<td>fneg[]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC00003E</td>
<td></td>
<td></td>
<td>150</td>
<td>FP[R]</td>
<td>fnmadd[]</td>
</tr>
<tr>
<td>A</td>
<td>59 EC00003E</td>
<td></td>
<td></td>
<td>150</td>
<td>FP[R]</td>
<td>fnmadds[]</td>
</tr>
</tbody>
</table>
### Power ISA Instruction Set Sorted by Category

#### Format

<table>
<thead>
<tr>
<th>Mode Dep.</th>
<th>Privilege</th>
<th>Category</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fnmsub</td>
<td>Floating Negative Multiply-Subtract</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fnmsubs</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fres</td>
<td>Floating Reciprocal Estimate Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>frsplt</td>
<td>Floating Round to Single-Precision</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>frsqrte</td>
<td>Floating Reciprocal Square Root Estimate</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fse</td>
<td>Floating Select</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fsqrt</td>
<td>Floating Square Root</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fsubs</td>
<td>Floating Subtract Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>mfsb0</td>
<td>Move To FPSCR Bit 0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>mfsb1</td>
<td>Move To FPSCR Bit 1</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>mffs0</td>
<td>Move To FPSCR Fields</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>mtfsf0</td>
<td>Move To FPSCR Field Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>mtfsfi</td>
<td>Move To FPSCR Field Immediate</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>fre</td>
<td>Floating Reciprocal Estimate Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>frin</td>
<td>Floating Round To Integer Nearest</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>frsip</td>
<td>Floating Round To Integer Plus</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>friz</td>
<td>Floating Round To Integer toward Zero</td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Set

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Category</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 FC00003C</td>
<td>FP[R]</td>
<td>fnmsub</td>
<td>Floating Negative Multiply-Subtract</td>
</tr>
<tr>
<td>59 EC00003C</td>
<td>FP[R]</td>
<td>fnmsubs</td>
<td>Floating Negative Multiply-Subtract Single</td>
</tr>
<tr>
<td>59 EC000030</td>
<td>FP[R]</td>
<td>fres</td>
<td>Floating Reciprocal Estimate Single</td>
</tr>
<tr>
<td>63 FC000018</td>
<td>FP[R]</td>
<td>frsplt</td>
<td>Floating Round to Single-Precision</td>
</tr>
<tr>
<td>63 FC000034</td>
<td>FP[R]</td>
<td>frsqrte</td>
<td>Floating Reciprocal Square Root Estimate</td>
</tr>
<tr>
<td>63 FC00002E</td>
<td>FP[R]</td>
<td>fse</td>
<td>Floating Select</td>
</tr>
<tr>
<td>63 FC00002C</td>
<td>FP[R]</td>
<td>fsqrt</td>
<td>Floating Square Root</td>
</tr>
<tr>
<td>63 FC000028</td>
<td>FP[R]</td>
<td>fsubs</td>
<td>Floating Subtract Single</td>
</tr>
<tr>
<td>63 FC00048E</td>
<td>FP[R]</td>
<td>mfsb0</td>
<td>Move To FPSCR Bit 0</td>
</tr>
<tr>
<td>63 FC00008C</td>
<td>FP[R]</td>
<td>mfsb1</td>
<td>Move To FPSCR Bit 1</td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>mffs0</td>
<td>Move To FPSCR Fields</td>
</tr>
<tr>
<td>A</td>
<td>FP[R]</td>
<td>mtfsf0</td>
<td>Move To FPSCR Field Immediate</td>
</tr>
<tr>
<td>63 FC000030</td>
<td>FP[R]</td>
<td>fre</td>
<td>Floating Reciprocal Estimate Single</td>
</tr>
<tr>
<td>63 FC000310</td>
<td>FP[R]</td>
<td>frin</td>
<td>Floating Round To Integer Nearest</td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>frzip</td>
<td>Floating Round To Integer Plus</td>
</tr>
<tr>
<td>X</td>
<td>FP[R]</td>
<td>friz</td>
<td>Floating Round To Integer toward Zero</td>
</tr>
</tbody>
</table>

#### Instruction Formats

- **LMA**
  - Multiply Accumulate Cross Halfword to Word
  - Multiply Accumulate High Halfword to Word
  - Multiply Accumulate Low Halfword to Word
- **Modulo**
  - Signed
  - Unsigned
- **Saturate**
  - Signed
  - Unsigned
- **Record OV**
  - Signed
  - Unsigned
<table>
<thead>
<tr>
<th>Format</th>
<th>Opcode</th>
<th>Mode Dep.¹</th>
<th>Privilege¹</th>
<th>Page</th>
<th>Category¹</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>XO</td>
<td>4</td>
<td>10000358</td>
<td>LMA</td>
<td>679</td>
<td></td>
<td>maclhwo[.]</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100003D8</td>
<td>LMA</td>
<td>679</td>
<td></td>
<td>maclhws[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100003D8</td>
<td>LMA</td>
<td>679</td>
<td></td>
<td>maclhwsos.</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000398</td>
<td>LMA</td>
<td>680</td>
<td></td>
<td>maclhwsu[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000398</td>
<td>LMA</td>
<td>680</td>
<td></td>
<td>maclhwsuos.</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000318</td>
<td>LMA</td>
<td>680</td>
<td></td>
<td>maclhwu[.]</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000318</td>
<td>LMA</td>
<td>680</td>
<td></td>
<td>maclhwusos.</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Unsigned &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000150</td>
<td>LMA</td>
<td>680</td>
<td></td>
<td>mulchw[.]</td>
<td>Multiply Cross Halfword to Word Signed</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000110</td>
<td>LMA</td>
<td>680</td>
<td></td>
<td>mulchwu[.]</td>
<td>Multiply Cross Halfword to Word Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000050</td>
<td>LMA</td>
<td>681</td>
<td></td>
<td>mulihw[.]</td>
<td>Multiply High Halfword to Word Signed</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000010</td>
<td>LMA</td>
<td>681</td>
<td></td>
<td>mulihwu[.]</td>
<td>Multiply High Halfword to Word Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000350</td>
<td>LMA</td>
<td>681</td>
<td></td>
<td>mulihw[]</td>
<td>Multiply Low Halfword to Word Signed</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000310</td>
<td>LMA</td>
<td>681</td>
<td></td>
<td>mulihwu[]</td>
<td>Multiply Low Halfword to Word Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000015C</td>
<td>LMA</td>
<td>682</td>
<td></td>
<td>nmacchw[.]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000015C</td>
<td>LMA</td>
<td>682</td>
<td></td>
<td>nmacchwo[.]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100001DC</td>
<td>LMA</td>
<td>682</td>
<td></td>
<td>nmacchws[.]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100001DC</td>
<td>LMA</td>
<td>682</td>
<td></td>
<td>nmacchwsos.</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000005C</td>
<td>LMA</td>
<td>683</td>
<td></td>
<td>nmachhw[.]</td>
<td>Negative Multiply Accumulate High Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000005C</td>
<td>LMA</td>
<td>683</td>
<td></td>
<td>nmachhwo[.]</td>
<td>Negative Multiply Accumulate High Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100000DC</td>
<td>LMA</td>
<td>683</td>
<td></td>
<td>nmachhws[.]</td>
<td>Negative Multiply Accumulate High Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100000DC</td>
<td>LMA</td>
<td>683</td>
<td></td>
<td>nmachhwsos.</td>
<td>Negative Multiply Accumulate High Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000035C</td>
<td>LMA</td>
<td>684</td>
<td></td>
<td>nmachhw[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000035C</td>
<td>LMA</td>
<td>684</td>
<td></td>
<td>nmachhwo[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100003DC</td>
<td>LMA</td>
<td>684</td>
<td></td>
<td>nmachhws[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100003DC</td>
<td>LMA</td>
<td>684</td>
<td></td>
<td>nmachhwsos.</td>
<td>Negative Multiply Accumulate Low Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00009C</td>
<td>LMA</td>
<td>673</td>
<td></td>
<td>dlmzb[.]</td>
<td>Determine Leftmost Zero Byte</td>
</tr>
<tr>
<td>DQ</td>
<td>56</td>
<td>E0000000</td>
<td>P</td>
<td>58</td>
<td>LSQ</td>
<td>liq</td>
<td>Load Quadword</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000228</td>
<td>LSQ</td>
<td>784</td>
<td>liqarx</td>
<td>Load Quadword And Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>62</td>
<td>F8000000</td>
<td>P</td>
<td>58</td>
<td>LSQ</td>
<td>stq</td>
<td>Store Quadword</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00016D</td>
<td>LSQ</td>
<td>785</td>
<td>stqcx.</td>
<td>Store Quadword Conditional Indexed and record CR0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0004AA</td>
<td>MA</td>
<td>63</td>
<td>lswi</td>
<td>Load String Word Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00042A</td>
<td>MA</td>
<td>63</td>
<td>lswx</td>
<td>Load String Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0005AA</td>
<td>MA</td>
<td>64</td>
<td>stswi</td>
<td>Store String Word Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00052A</td>
<td>MA</td>
<td>64</td>
<td>stswx</td>
<td>Store String Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00035C</td>
<td>S</td>
<td>44</td>
<td>clrbhrb</td>
<td>Clear BHRB</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000324</td>
<td>H</td>
<td>867</td>
<td>doze</td>
<td>Doze</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0006AC</td>
<td>S</td>
<td>790</td>
<td>einio</td>
<td>Enforce In-order Execution of I/O</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Primary</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000224</td>
<td>H</td>
<td>865</td>
<td>S</td>
<td>hrfid</td>
<td>Return From Interrupt Doubleword Hypervisor</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0006AA</td>
<td>H</td>
<td>876</td>
<td>S</td>
<td>lbzcix</td>
<td>Load Byte and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0006EA</td>
<td>H</td>
<td>876</td>
<td>S</td>
<td>ldcix</td>
<td>Load Doubleword Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00066A</td>
<td>H</td>
<td>876</td>
<td>S</td>
<td>lhzcix</td>
<td>Load Halfword and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00062A</td>
<td>H</td>
<td>876</td>
<td>S</td>
<td>lwzcix</td>
<td>Load Word and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>XFX</td>
<td>31</td>
<td>7C00025C</td>
<td>44</td>
<td>S</td>
<td>mfbhrbe</td>
<td>Move From Branch History Rolling Buffer</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0004A6</td>
<td>32 P</td>
<td>927</td>
<td>S</td>
<td>mfsr</td>
<td>Move From Segment Register</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000526</td>
<td>32 P</td>
<td>927</td>
<td>S</td>
<td>mfsrin</td>
<td>Move From Segment Register Indirect</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000164</td>
<td>P</td>
<td>886</td>
<td>S</td>
<td>mtmsrd</td>
<td>Move To Machine State Register Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000126</td>
<td>108</td>
<td>S</td>
<td>mtsie</td>
<td>Move To Split Little Endian</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0001A4</td>
<td>32 P</td>
<td>926</td>
<td>S</td>
<td>mtsr</td>
<td>Move To Segment Register</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0001E4</td>
<td>32 P</td>
<td>926</td>
<td>S</td>
<td>mtsrin</td>
<td>Move To Segment Register Indirect</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00007A</td>
<td>SR P</td>
<td>923</td>
<td>S</td>
<td>silbfee.</td>
<td>SLB Find Entry ESID</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0003E4</td>
<td>P</td>
<td>920</td>
<td>S</td>
<td>silbie</td>
<td>SLB Invalidate All</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000364</td>
<td>P</td>
<td>919</td>
<td>S</td>
<td>silbie</td>
<td>SLB Invalidate Entry</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000726</td>
<td>P</td>
<td>923</td>
<td>S</td>
<td>slbmfee</td>
<td>SLB Move From Entry ESID</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0006A6</td>
<td>P</td>
<td>922</td>
<td>S</td>
<td>slbmfev</td>
<td>SLB Move From Entry VSID</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000324</td>
<td>P</td>
<td>921</td>
<td>S</td>
<td>slbmte</td>
<td>SLB Move To Entry</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000364</td>
<td>H</td>
<td>867</td>
<td>S</td>
<td>sleep</td>
<td>Sleep</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0007AA</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stbcix</td>
<td>Store Byte Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0007EA</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stdcix</td>
<td>Store Doubleword Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00076A</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>sthcix</td>
<td>Store Halfword and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00072A</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stwcix</td>
<td>Store Word and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00022E</td>
<td>64 P</td>
<td>931</td>
<td>S</td>
<td>tlibiel</td>
<td>TLB Invalidate Entry Local</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000224</td>
<td>64 P</td>
<td>931</td>
<td>S</td>
<td>tlibiel</td>
<td>TLB Invalidate Entry</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00007A</td>
<td>SR P</td>
<td>923</td>
<td>S</td>
<td>tlbia</td>
<td>TLB Invalidate All</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000264</td>
<td>64 H</td>
<td>928</td>
<td>S</td>
<td>tlibie</td>
<td>TLB Invalidate Entry</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000224</td>
<td>64 P</td>
<td>931</td>
<td>S</td>
<td>tlibiel</td>
<td>TLB Invalidate Entry Local</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0002E6</td>
<td>813</td>
<td>S.out</td>
<td>mftb</td>
<td>Move From Time Base</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000015</td>
<td>P</td>
<td>1007</td>
<td>S.PC</td>
<td>msgclrp</td>
<td>Message Clear Privileged</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000011</td>
<td>P</td>
<td>1007</td>
<td>S.PC</td>
<td>msgsndp</td>
<td>Message Send Privileged</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0001DC</td>
<td>H</td>
<td>1006</td>
<td>S.PC</td>
<td>msgclrp</td>
<td>Message Clear</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00019C</td>
<td>H</td>
<td>1006</td>
<td>S.PC</td>
<td>msgsnd</td>
<td>Message Send</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0000A6</td>
<td>P</td>
<td>888</td>
<td>S.E</td>
<td>mfmsr</td>
<td>Move From Machine State Register</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000124</td>
<td>P</td>
<td>884</td>
<td>S.E</td>
<td>mtmsr</td>
<td>Move To Machine State Register</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00046C</td>
<td>H.PH</td>
<td>934</td>
<td>S.E</td>
<td>tlbinc</td>
<td>TLB Synchronize</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000020</td>
<td>F</td>
<td>594</td>
<td>SP</td>
<td>brinc</td>
<td>Bit Reversed Increment</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000020</td>
<td>B</td>
<td>594</td>
<td>SP</td>
<td>evabs</td>
<td>Vector Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000020</td>
<td>2</td>
<td>594</td>
<td>SP</td>
<td>evaddi</td>
<td>Vector Add Immediate Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000049</td>
<td>C</td>
<td>594</td>
<td>SP</td>
<td>evaddsmiaaw</td>
<td>Vector Add Signed, Modulo, Integer to Accumulator Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000041</td>
<td>C</td>
<td>595</td>
<td>SP</td>
<td>evaddssiaaw</td>
<td>Vector Add Signed, Saturate, Integer to Accumulator Word</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004C8</td>
<td>595 SP</td>
<td></td>
<td></td>
<td>evaddumiaaw Vector Add Unsigned, Modulo, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004C0</td>
<td>595 SP</td>
<td></td>
<td></td>
<td>evaddusiaaw Vector Add Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000200</td>
<td>595 SP</td>
<td></td>
<td></td>
<td>evaddw Vector Add Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000211</td>
<td>596 SP</td>
<td></td>
<td></td>
<td>evand Vector AND</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000212</td>
<td>596 SP</td>
<td></td>
<td></td>
<td>evandc Vector AND with Complement</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000234</td>
<td>596 SP</td>
<td></td>
<td></td>
<td>evcmpeq Vector Compare Equal</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000231</td>
<td>596 SP</td>
<td></td>
<td></td>
<td>evcmpeq Vector Compare Greater Than Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000230</td>
<td>597 SP</td>
<td></td>
<td></td>
<td>evcmpeqtu Vector Compare Greater Than Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000233</td>
<td>597 SP</td>
<td></td>
<td></td>
<td>evcmpeqlt Vector Compare Less Than Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000232</td>
<td>597 SP</td>
<td></td>
<td></td>
<td>evcmpeqltu Vector Compare Less Than Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020E</td>
<td>598 SP</td>
<td></td>
<td></td>
<td>evcntlsw Vector Count Leading Signed Bits Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020D</td>
<td>598 SP</td>
<td></td>
<td></td>
<td>evcntlzsw Vector Count Leading Zeros Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004C6</td>
<td>598 SP</td>
<td></td>
<td></td>
<td>evdivws Vector Divide Word Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004C7</td>
<td>599 SP</td>
<td></td>
<td></td>
<td>evdivuw Vector Divide Word Unsaturated</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000219</td>
<td>599 SP</td>
<td></td>
<td></td>
<td>eveqv Vector Equivalent</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020A</td>
<td>599 SP</td>
<td></td>
<td></td>
<td>evextsb Vector Extend Sign Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020B</td>
<td>599 SP</td>
<td></td>
<td></td>
<td>evextsh Vector Extend Sign Half Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000301</td>
<td>600 SP</td>
<td></td>
<td></td>
<td>evldd Vector Load Double Word into Double Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000300</td>
<td>600 SP</td>
<td></td>
<td></td>
<td>evlddx Vector Load Double Word into Double Word Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000305</td>
<td>600 SP</td>
<td></td>
<td></td>
<td>evldh Vector Load Double into Four Half Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000304</td>
<td>600 SP</td>
<td></td>
<td></td>
<td>evldhx Vector Load Double into Four Half Words Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000303</td>
<td>601 SP</td>
<td></td>
<td></td>
<td>evldw Vector Load Double into Two Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000302</td>
<td>601 SP</td>
<td></td>
<td></td>
<td>evldwx Vector Load Double into Two Words Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000309</td>
<td>601 SP</td>
<td></td>
<td></td>
<td>evlhhesplat Vector Load Half Word into Half Words Even and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000308</td>
<td>601 SP</td>
<td></td>
<td></td>
<td>evlhhesplatx Vector Load Half Word into Half Words Even and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030F</td>
<td>602 SP</td>
<td></td>
<td></td>
<td>evlhhosplat Vector Load Half Word into Half Word Odd Signed and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030E</td>
<td>602 SP</td>
<td></td>
<td></td>
<td>evlhhosplax Vector Load Half Word into Half Word Odd Signed and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030D</td>
<td>602 SP</td>
<td></td>
<td></td>
<td>evlhousplat Vector Load Half Word into Half Word Odd Unsigned and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030C</td>
<td>602 SP</td>
<td></td>
<td></td>
<td>evlhousplax Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000311</td>
<td>603 SP</td>
<td></td>
<td></td>
<td>evlwhe Vector Load Word into Two Half Words Even</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000310</td>
<td>603 SP</td>
<td></td>
<td></td>
<td>evlwhex Vector Load Word into Two Half Words Even Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000317</td>
<td>603 SP</td>
<td></td>
<td></td>
<td>evlwhos Vector Load Word into Two Half Words Odd Signed (with sign extension)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000316</td>
<td>603 SP</td>
<td></td>
<td></td>
<td>evlwhosx Vector Load Word into Two Half Words Odd Signed Indexed (with sign extension)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000315</td>
<td>604 SP</td>
<td></td>
<td></td>
<td>evlwhou Vector Load Word into Two Half Words Odd Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000314</td>
<td>604 SP</td>
<td></td>
<td></td>
<td>evlwhoux Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000031D</td>
<td>604 SP</td>
<td></td>
<td></td>
<td>evlwhsplat Vector Load Word into Two Half Words and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000031C</td>
<td>604 SP</td>
<td></td>
<td></td>
<td>evlwhsplatx Vector Load Word into Two Half Words and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000319</td>
<td>605 SP</td>
<td></td>
<td></td>
<td>evlwssplat Vector Load Word into Word and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000318</td>
<td>605 SP</td>
<td></td>
<td></td>
<td>evlwssplax Vector Load Word into Word and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Primary</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022C</td>
<td></td>
<td>605</td>
<td>SP</td>
<td>evmergehi</td>
<td>Vector Merge High</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022E</td>
<td></td>
<td>606</td>
<td>SP</td>
<td>evmergehilo</td>
<td>Vector Merge High/Low</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022D</td>
<td></td>
<td>605</td>
<td>SP</td>
<td>evmergelohi</td>
<td>Vector Merge Low</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022F</td>
<td></td>
<td>606</td>
<td>SP</td>
<td>evmergelohi</td>
<td>Vector Merge Low/Low</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052B</td>
<td></td>
<td>606</td>
<td>SP</td>
<td>evmhegsmfaa</td>
<td>Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AB</td>
<td></td>
<td>606</td>
<td>SP</td>
<td>evmhegsmfan</td>
<td>Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000529</td>
<td></td>
<td>607</td>
<td>SP</td>
<td>evmhegsniaa</td>
<td>Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005A9</td>
<td></td>
<td>607</td>
<td>SP</td>
<td>evmhegsmian</td>
<td>Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052B</td>
<td></td>
<td>607</td>
<td>SP</td>
<td>evmhegumiaa</td>
<td>Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005A8</td>
<td></td>
<td>607</td>
<td>SP</td>
<td>evmhegumian</td>
<td>Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000040B</td>
<td></td>
<td>608</td>
<td>SP</td>
<td>evmhesmf</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000042B</td>
<td></td>
<td>608</td>
<td>SP</td>
<td>evmhesmfa</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000050B</td>
<td></td>
<td>608</td>
<td>SP</td>
<td>evmhesmfaaw</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000058B</td>
<td></td>
<td>608</td>
<td>SP</td>
<td>evmhesmfanw</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000409</td>
<td></td>
<td>609</td>
<td>SP</td>
<td>evmhesmi</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000429</td>
<td></td>
<td>609</td>
<td>SP</td>
<td>evmhesmia</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000509</td>
<td></td>
<td>609</td>
<td>SP</td>
<td>evmhesmiaaw</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000589</td>
<td></td>
<td>609</td>
<td>SP</td>
<td>evmhesmianw</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000403</td>
<td></td>
<td>610</td>
<td>SP</td>
<td>evmhessf</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000423</td>
<td></td>
<td>610</td>
<td>SP</td>
<td>evmhessfa</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000503</td>
<td></td>
<td>611</td>
<td>SP</td>
<td>evmhesffaaw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000583</td>
<td></td>
<td>611</td>
<td>SP</td>
<td>evmhesfanw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000501</td>
<td></td>
<td>612</td>
<td>SP</td>
<td>evmheesiaaw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000581</td>
<td></td>
<td>612</td>
<td>SP</td>
<td>evmheesianw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000408</td>
<td></td>
<td>613</td>
<td>SP</td>
<td>evmheumi</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000428</td>
<td></td>
<td>613</td>
<td>SP</td>
<td>evmheumia</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000508</td>
<td></td>
<td>613</td>
<td>SP</td>
<td>evmheumiaaw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000588</td>
<td>613</td>
<td>SP</td>
<td>evmheumianw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000500</td>
<td>614</td>
<td>SP</td>
<td>evmheusiaaw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000580</td>
<td>614</td>
<td>SP</td>
<td>evmheusianw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052F</td>
<td>615</td>
<td>SP</td>
<td>evmhogsmaaa</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AF</td>
<td>615</td>
<td>SP</td>
<td>evmhogsman</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052D</td>
<td>615</td>
<td>SP</td>
<td>evmhogsmaia</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AD</td>
<td>615</td>
<td>SP</td>
<td>evmhogsman</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052C</td>
<td>616</td>
<td>SP</td>
<td>evmhogumiaa</td>
<td>Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AC</td>
<td>616</td>
<td>SP</td>
<td>evmhogumian</td>
<td>Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000040F</td>
<td>616</td>
<td>SP</td>
<td>evmhosmfa</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000042F</td>
<td>616</td>
<td>SP</td>
<td>evmhosmfaa</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000050F</td>
<td>617</td>
<td>SP</td>
<td>evmhosmfaaw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000058F</td>
<td>617</td>
<td>SP</td>
<td>evmhosmfanw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000040D</td>
<td>617</td>
<td>SP</td>
<td>evmhosmi</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000042D</td>
<td>617</td>
<td>SP</td>
<td>evmhosmia</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000050D</td>
<td>618</td>
<td>SP</td>
<td>evmhosmiaaw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000058D</td>
<td>617</td>
<td>SP</td>
<td>evmhosmianw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000407</td>
<td>619</td>
<td>SP</td>
<td>evmhossf</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000427</td>
<td>619</td>
<td>SP</td>
<td>evmhossfa</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000507</td>
<td>620</td>
<td>SP</td>
<td>evmhossfaaw</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000587</td>
<td>620</td>
<td>SP</td>
<td>evmhossfanw</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000505</td>
<td>621</td>
<td>SP</td>
<td>evmhossiaaw</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000585</td>
<td>621</td>
<td>SP</td>
<td>evmhossianw</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Mode Dep.¹</td>
<td>Privilege</td>
<td>Page</td>
<td>Category¹</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-----------</td>
<td>------</td>
<td>-----------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>621</td>
<td>SP</td>
<td>evmhouni</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>621</td>
<td>SP</td>
<td>evmhounia</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>622</td>
<td>SP</td>
<td>evmhoumiaaaw</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>618</td>
<td>SP</td>
<td>evmhoumianw</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Modulo and Accumulate Negative into Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>622</td>
<td>SP</td>
<td>evmhousiaaw</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>623</td>
<td>SP</td>
<td>evmra</td>
<td>Initialize Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>623</td>
<td>SP</td>
<td>evmwhsmf</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>623</td>
<td>SP</td>
<td>evmwhsmfa</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>623</td>
<td>SP</td>
<td>evmwhsmi</td>
<td>Vector Multiply Word High Signed, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>624</td>
<td>SP</td>
<td>evmwhssf</td>
<td>Vector Multiply Word High Signed, Saturate, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>624</td>
<td>SP</td>
<td>evmwhssfa</td>
<td>Vector Multiply Word High Signed, Saturate, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>624</td>
<td>SP</td>
<td>evmwhumi</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>624</td>
<td>SP</td>
<td>evmwhumia</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>625</td>
<td>SP</td>
<td>evmwismiaaw</td>
<td>Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>625</td>
<td>SP</td>
<td>evmwismianw</td>
<td>Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>625</td>
<td>SP</td>
<td>evmwllssiaaw</td>
<td>Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>625</td>
<td>SP</td>
<td>evmwllssianw</td>
<td>Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>626</td>
<td>SP</td>
<td>evmwulumi</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>626</td>
<td>SP</td>
<td>evmwulumia</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>626</td>
<td>SP</td>
<td>evmwulumiaaw</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>626</td>
<td>SP</td>
<td>evmwllumianw</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>627</td>
<td>SP</td>
<td>evmwlusiaaw</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>627</td>
<td>SP</td>
<td>evmwlusianw</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>627</td>
<td>SP</td>
<td>evmwmsmf</td>
<td>Vector Multiply Word Signed, Modulo, Fractional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>627</td>
<td>SP</td>
<td>evmwmsfa</td>
<td>Vector Multiply Word Signed, Modulo, Fractional to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>628</td>
<td>SP</td>
<td>evmwmsfaa</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005DB</td>
<td>SP</td>
<td>628</td>
<td>SP</td>
<td>evmwsfan</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000459</td>
<td>SP</td>
<td>628</td>
<td>SP</td>
<td>evmwsmi</td>
<td>Vector Multiply Word Signed, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000479</td>
<td>SP</td>
<td>628</td>
<td>SP</td>
<td>evmwsmia</td>
<td>Vector Multiply Word Signed, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000559</td>
<td>SP</td>
<td>628</td>
<td>SP</td>
<td>evmwsmiaa</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D9</td>
<td>SP</td>
<td>628</td>
<td>SP</td>
<td>evmwsmian</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000453</td>
<td>SP</td>
<td>629</td>
<td>SP</td>
<td>evmwssf</td>
<td>Vector Multiply Word Signed, Saturate, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000473</td>
<td>SP</td>
<td>629</td>
<td>SP</td>
<td>evmwssfa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000553</td>
<td>SP</td>
<td>629</td>
<td>SP</td>
<td>evmwssfaa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D3</td>
<td>SP</td>
<td>630</td>
<td>SP</td>
<td>evmwssfan</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D8</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evmwumia</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000478</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evmwumiaa</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000558</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evmwumiaa</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D8</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evmwumian</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000021E</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evnand</td>
<td>Vector NAND</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000209</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evneg</td>
<td>Vector Negate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000218</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evnor</td>
<td>Vector NOR</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000217</td>
<td>SP</td>
<td>632</td>
<td>SP</td>
<td>evor</td>
<td>Vector OR</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000021B</td>
<td>SP</td>
<td>632</td>
<td>SP</td>
<td>evorc</td>
<td>Vector OR with Complement</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000228</td>
<td>SP</td>
<td>632</td>
<td>SP</td>
<td>evrlw</td>
<td>Vector Rotate Left Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022A</td>
<td>SP</td>
<td>633</td>
<td>SP</td>
<td>evrlwi</td>
<td>Vector Rotate Left Word Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020C</td>
<td>SP</td>
<td>633</td>
<td>SP</td>
<td>evmdw</td>
<td>Vector Round Word</td>
</tr>
<tr>
<td>EVS</td>
<td>4</td>
<td>10000278</td>
<td>SP</td>
<td>633</td>
<td>SP</td>
<td>evsel</td>
<td>Vector Select</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000224</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsiw</td>
<td>Vector Shift Left Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000226</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsiwi</td>
<td>Vector Shift Left Word Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022B</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsplatf</td>
<td>Vector Splat Fractional Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000229</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsplat</td>
<td>Vector Splat Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000223</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsrsis</td>
<td>Vector Shift Right Word Immediate Signed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000222</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsrsiwu</td>
<td>Vector Shift Right Word Immediate Unsigned</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000221</td>
<td>SP</td>
<td>635</td>
<td>SP</td>
<td>evsrsuw</td>
<td>Vector Shift Right Word Signed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000220</td>
<td>SP</td>
<td>635</td>
<td>SP</td>
<td>evsrsuwu</td>
<td>Vector Shift Right Word Unsigned</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000321</td>
<td>SP</td>
<td>635</td>
<td>SP</td>
<td>evstd</td>
<td>Vector Store Double of Double</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000320</td>
<td>SP</td>
<td>635</td>
<td>SP</td>
<td>evstdx</td>
<td>Vector Store Double of Double Indexed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000325</td>
<td>SP</td>
<td>636</td>
<td>SP</td>
<td>evstdh</td>
<td>Vector Store Double of Four Half Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000324</td>
<td>SP</td>
<td>636</td>
<td>SP</td>
<td>evstdhx</td>
<td>Vector Store Double of Four Half Words Indexed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000323</td>
<td>SP</td>
<td>636</td>
<td>SP</td>
<td>evstdw</td>
<td>Vector Store Double of Two Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000322</td>
<td>SP</td>
<td>636</td>
<td>SP</td>
<td>evstdwx</td>
<td>Vector Store Double of Two Words Indexed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000331</td>
<td>SP</td>
<td>637</td>
<td>SP</td>
<td>evstwhe</td>
<td>Vector Store Word of Two Half Words from Even</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000330</td>
<td>SP</td>
<td>637</td>
<td>SP</td>
<td>evstwhex</td>
<td>Vector Store Word of Two Half Words from Even Indexed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000335</td>
<td>SP</td>
<td>637</td>
<td>SP</td>
<td>evstwho</td>
<td>Vector Store Word of Two Half Words from Odd</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000334</td>
<td>SP</td>
<td>637</td>
<td>SP</td>
<td>evstwhox</td>
<td>Vector Store Word of Two Half Words from Odd Indexed</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000339</td>
<td>SP</td>
<td>637</td>
<td>SP</td>
<td>evstwwe</td>
<td>Vector Store Word of Word from Even</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000338</td>
<td>SP</td>
<td>637</td>
<td>SP</td>
<td>evstwwex</td>
<td>Vector Store Word of Word from Even Indexed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000033D</td>
<td>SP</td>
<td>638</td>
<td>SP</td>
<td>evstwwo</td>
<td>Vector Store Word of Word from Odd</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000033C</td>
<td>SP</td>
<td>638</td>
<td>SP</td>
<td>evstwwox</td>
<td>Vector Store Word of Word from Odd Indexed</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004CB</td>
<td>SP</td>
<td>638</td>
<td>SP</td>
<td>evsubfsmiaaw</td>
<td>Vector Subtract Signed, Modulo, Integer to Accumulator Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004C3</td>
<td>SP</td>
<td>638</td>
<td>SP</td>
<td>evsubfssiaaw</td>
<td>Vector Subtract Signed, Saturate, Integer to Accumulator Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004CA</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evsubfumiaaw</td>
<td>Vector Subtract Unsigned, Modulo, Integer to Accumulator Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100004C2</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evsubfusiaaw</td>
<td>Vector Subtract Unsigned, Saturate, Integer to Accumulator Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000204</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evsubfw</td>
<td>Vector Subtract from Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000206</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evsubifw</td>
<td>Vector Subtract Immediate from Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000216</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evxor</td>
<td>Vector XOR</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E4</td>
<td>SP.FD</td>
<td>660</td>
<td>SP.FD</td>
<td>efdabs</td>
<td>Floating-Point Double-Precision Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E0</td>
<td>SP.FD</td>
<td>661</td>
<td>SP.FD</td>
<td>efdadd</td>
<td>Floating-Point Double-Precision Add</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002EF</td>
<td>SP.FD</td>
<td>666</td>
<td>SP.FD</td>
<td>efdcs</td>
<td>Floating-Point Double-Precision Convert from Single-Precision</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F3</td>
<td>SP.FD</td>
<td>664</td>
<td>SP.FD</td>
<td>efdcsfs</td>
<td>Convert Floating-Point Double-Precision from Signed Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F1</td>
<td>SP.FD</td>
<td>663</td>
<td>SP.FD</td>
<td>efdcsfi</td>
<td>Convert Floating-Point Double-Precision from Signed Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E3</td>
<td>SP.FD</td>
<td>664</td>
<td>SP.FD</td>
<td>efdcsid</td>
<td>Convert Floating-Point Double-Precision from Signed Integer Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F2</td>
<td>SP.FD</td>
<td>664</td>
<td>SP.FD</td>
<td>efdcfuf</td>
<td>Convert Floating-Point Double-Precision from Unsigned Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F0</td>
<td>SP.FD</td>
<td>663</td>
<td>SP.FD</td>
<td>efdcfui</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E2</td>
<td>SP.FD</td>
<td>664</td>
<td>SP.FD</td>
<td>efdcfuid</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002EE</td>
<td>SP.FD</td>
<td>662</td>
<td>SP.FD</td>
<td>efdcmpq</td>
<td>Floating-Point Double-Precision Compare Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002EC</td>
<td>SP.FD</td>
<td>662</td>
<td>SP.FD</td>
<td>efdcmpgt</td>
<td>Floating-Point Double-Precision Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002ED</td>
<td>SP.FD</td>
<td>662</td>
<td>SP.FD</td>
<td>efdcmplt</td>
<td>Floating-Point Double-Precision Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F7</td>
<td>SP.FD</td>
<td>666</td>
<td>SP.FD</td>
<td>efdfcstf</td>
<td>Convert Floating-Point Double-Precision to Signed Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F5</td>
<td>SP.FD</td>
<td>664</td>
<td>SP.FD</td>
<td>efdfcsd</td>
<td>Convert Floating-Point Double-Precision to Signed Integer Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002EB</td>
<td>SP.FD</td>
<td>665</td>
<td>SP.FD</td>
<td>efdfctsidz</td>
<td>Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002FA</td>
<td>SP.FD</td>
<td>666</td>
<td>SP.FD</td>
<td>efdfctsz</td>
<td>Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F6</td>
<td>SP.FD</td>
<td>666</td>
<td>SP.FD</td>
<td>efdfctuf</td>
<td>Convert Floating-Point Double-Precision to Unsigned Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F4</td>
<td>SP.FD</td>
<td>664</td>
<td>SP.FD</td>
<td>efdfctui</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002EA</td>
<td>SP.FD</td>
<td>665</td>
<td>SP.FD</td>
<td>efdfctuidz</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002F8</td>
<td>SP.FD</td>
<td>666</td>
<td>SP.FD</td>
<td>efdfctuiz</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E9</td>
<td>SP.FD</td>
<td>661</td>
<td>SP.FD</td>
<td>efdfdiv</td>
<td>Floating-Point Double-Precision Divide</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E8</td>
<td>SP.FD</td>
<td>661</td>
<td>SP.FD</td>
<td>efdfmul</td>
<td>Floating-Point Double-Precision Multiply</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E5</td>
<td></td>
<td>660</td>
<td>SP.FD</td>
<td>efdnabs</td>
<td>Floating-Point Double-Precision Negative Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E6</td>
<td></td>
<td>660</td>
<td>SP.FD</td>
<td>efdneg</td>
<td>Floating-Point Double-Precision Negate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002E1</td>
<td></td>
<td>661</td>
<td>SP.FD</td>
<td>efsadd</td>
<td>Floating-Point Double-Precision Add</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002FE</td>
<td></td>
<td>663</td>
<td>SP.FD</td>
<td>eftsteq</td>
<td>Floating-Point Double-Precision Test Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002FC</td>
<td></td>
<td>662</td>
<td>SP.FD</td>
<td>eftstgt</td>
<td>Floating-Point Double-Precision Test Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002FD</td>
<td></td>
<td>663</td>
<td>SP.FD</td>
<td>efdsz</td>
<td>Floating-Point Double-Precision Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>667</td>
<td>SP.FD</td>
<td>efsz</td>
<td>Floating-Point Single-Precision Compare Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C4</td>
<td></td>
<td>653</td>
<td>SP.FS</td>
<td>efsabs</td>
<td>Floating-Point Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C0</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efsadd</td>
<td>Floating-Point Add</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D3</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmpeg</td>
<td>Floating-Point Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D1</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmplt</td>
<td>Floating-Point Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D2</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmpeq</td>
<td>Floating-Point Compare Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D0</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efsbpeq</td>
<td>Floating-Point Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efsbplt</td>
<td>Floating-Point Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmpeq</td>
<td>Floating-Point Compare Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D7</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efsbpeq</td>
<td>Floating-Point Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D5</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efsbplt</td>
<td>Floating-Point Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D8</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D9</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C9</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmpeq</td>
<td>Floating-Point Compare Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C8</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C5</td>
<td></td>
<td>653</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C6</td>
<td></td>
<td>653</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C1</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002DE</td>
<td></td>
<td>657</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002DC</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002DD</td>
<td></td>
<td>657</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002DA</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D6</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D4</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002D8</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C9</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CA</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>657</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>657</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>657</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>654</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>655</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td></td>
<td>656</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td></td>
<td>657</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CE</td>
<td></td>
<td>658</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CF</td>
<td></td>
<td>659</td>
<td>SP.FS</td>
<td>efcmstz</td>
<td>Floating-Point Compare with Round toward Zero</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000296</td>
<td></td>
<td></td>
<td>652</td>
<td>SP.FV</td>
<td>evfscuf</td>
<td>Vector Convert Floating-Point to Unsigned Fraction</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000294</td>
<td></td>
<td></td>
<td>651</td>
<td>SP.FV</td>
<td>evfscui</td>
<td>Vector Convert Floating-Point to Unsigned Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000298</td>
<td></td>
<td></td>
<td>651</td>
<td>SP.FV</td>
<td>evfscuiz</td>
<td>Vector Convert Floating-Point to Unsigned Integer with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000289</td>
<td></td>
<td></td>
<td>646</td>
<td>SP.FV</td>
<td>evfsdiv</td>
<td>Vector Floating-Point Divide</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000288</td>
<td></td>
<td></td>
<td>646</td>
<td>SP.FV</td>
<td>evfsmul</td>
<td>Vector Floating-Point Multiply</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000285</td>
<td></td>
<td></td>
<td>645</td>
<td>SP.FV</td>
<td>evfsnabs</td>
<td>Vector Floating-Point Negative Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000286</td>
<td></td>
<td></td>
<td>645</td>
<td>SP.FV</td>
<td>evfsneg</td>
<td>Vector Floating-Point Negate</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000281</td>
<td></td>
<td></td>
<td>646</td>
<td>SP.FV</td>
<td>evfssub</td>
<td>Vector Floating-Point Subtract</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000029E</td>
<td></td>
<td></td>
<td>649</td>
<td>SP.FV</td>
<td>evfststeq</td>
<td>Vector Floating-Point Test Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000029C</td>
<td></td>
<td></td>
<td>648</td>
<td>SP.FV</td>
<td>evfststgt</td>
<td>Vector Floating-Point Test Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000029D</td>
<td></td>
<td></td>
<td>649</td>
<td>SP.FV</td>
<td>evfstslt</td>
<td>Vector Floating-Point Test Less Than</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00071D</td>
<td></td>
<td></td>
<td>808</td>
<td>TM</td>
<td>tabort</td>
<td>Transaction Abort</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00065D</td>
<td></td>
<td></td>
<td>809</td>
<td>TM</td>
<td>tabortdc</td>
<td>Transaction Abort Doubleword Conditional</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00066D</td>
<td></td>
<td></td>
<td>810</td>
<td>TM</td>
<td>tabortdci</td>
<td>Transaction Abort Doubleword Conditional Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00061D</td>
<td></td>
<td></td>
<td>809</td>
<td>TM</td>
<td>tabortwc</td>
<td>Transaction Abort Word Conditional</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00069D</td>
<td></td>
<td></td>
<td>809</td>
<td>TM</td>
<td>tabortwci</td>
<td>Transaction Abort Word Conditional Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00051D</td>
<td></td>
<td></td>
<td>806</td>
<td>TM</td>
<td>tbegin</td>
<td>Transaction Begin</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00059C</td>
<td></td>
<td></td>
<td>811</td>
<td>TM</td>
<td>tcheck</td>
<td>Transaction Check</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00055C</td>
<td></td>
<td></td>
<td>807</td>
<td>TM</td>
<td>tend</td>
<td>Transaction End</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007DD</td>
<td></td>
<td></td>
<td>880</td>
<td>TM</td>
<td>trechpkt</td>
<td>Transaction Recheckpoint</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00075D</td>
<td></td>
<td></td>
<td>879</td>
<td>TM</td>
<td>treclaim</td>
<td>Transaction Reclaim</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00056C</td>
<td></td>
<td></td>
<td>810</td>
<td>TM</td>
<td>tsr</td>
<td>Transaction Suspend or Resume</td>
</tr>
<tr>
<td>VX</td>
<td>60 F0000490</td>
<td></td>
<td></td>
<td>267</td>
<td>V</td>
<td>bcddadd</td>
<td>Decimal Add Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>60 F0000440</td>
<td></td>
<td></td>
<td>267</td>
<td>V</td>
<td>bcdsub</td>
<td>Decimal Subtract Modulo</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00005E</td>
<td></td>
<td></td>
<td>184</td>
<td>V</td>
<td>lvebx</td>
<td>Load Vector Element Byte Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00046E</td>
<td></td>
<td></td>
<td>181</td>
<td>V</td>
<td>lvehx</td>
<td>Load Vector Element Halfword Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00008E</td>
<td></td>
<td></td>
<td>181</td>
<td>V</td>
<td>lvewx</td>
<td>Load Vector Element Word Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00000C</td>
<td></td>
<td></td>
<td>186</td>
<td>V</td>
<td>lvsl</td>
<td>Load Vector for Shift Left</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00004C</td>
<td></td>
<td></td>
<td>186</td>
<td>V</td>
<td>lvsr</td>
<td>Load Vector for Shift Right</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000CE</td>
<td></td>
<td></td>
<td>182</td>
<td>V</td>
<td>lvx</td>
<td>Load Vector Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0002CE</td>
<td></td>
<td></td>
<td>182</td>
<td>V</td>
<td>lvxl</td>
<td>Load Vector Indexed Last</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000604</td>
<td></td>
<td></td>
<td>268</td>
<td>V</td>
<td>mfvscr</td>
<td>Move From Vector Status and Control Register</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000644</td>
<td></td>
<td></td>
<td>268</td>
<td>V</td>
<td>mtvscr</td>
<td>Move To Vector Status and Control Register</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00010E</td>
<td></td>
<td></td>
<td>184</td>
<td>V</td>
<td>stvebx</td>
<td>Store Vector Element Byte Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00014E</td>
<td></td>
<td></td>
<td>184</td>
<td>V</td>
<td>stvehx</td>
<td>Store Vector Element Halfword Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00018E</td>
<td></td>
<td></td>
<td>185</td>
<td>V</td>
<td>stvewx</td>
<td>Store Vector Element Word Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001CE</td>
<td></td>
<td></td>
<td>182</td>
<td>V</td>
<td>stvx</td>
<td>Store Vector Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003CE</td>
<td></td>
<td></td>
<td>185</td>
<td>V</td>
<td>stvxl</td>
<td>Store Vector Indexed Last</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000140</td>
<td></td>
<td></td>
<td>206</td>
<td>V</td>
<td>vaddcuq</td>
<td>Vector Add &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000180</td>
<td></td>
<td></td>
<td>202</td>
<td>V</td>
<td>vaddcuw</td>
<td>Vector Add and Write Carry-Out Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 1000003D</td>
<td></td>
<td></td>
<td>206</td>
<td>V</td>
<td>vaddecuq</td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>VA</td>
<td>4 1000003C</td>
<td></td>
<td></td>
<td>206</td>
<td>V</td>
<td>vaddeuqm</td>
<td>Vector Add Extended Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000000A</td>
<td></td>
<td></td>
<td>244</td>
<td>V</td>
<td>vaddfp</td>
<td>Vector Add Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000300</td>
<td></td>
<td></td>
<td>202</td>
<td>V</td>
<td>vaddsbs</td>
<td>Vector Add Signed Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000340</td>
<td></td>
<td></td>
<td>202</td>
<td>V</td>
<td>vaddshs</td>
<td>Vector Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000380</td>
<td></td>
<td></td>
<td>203</td>
<td>V</td>
<td>vaddsws</td>
<td>Vector Add Signed Word Saturate</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>------------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000000</td>
<td>V</td>
<td>203</td>
<td>V</td>
<td>Vector Add Unsigned Byte Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000200</td>
<td>V</td>
<td>205</td>
<td>V</td>
<td>Vector Add Unsigned Byte Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000000C</td>
<td>V</td>
<td>203</td>
<td>V</td>
<td>Vector Add Unsigned Doubleword Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000040</td>
<td>V</td>
<td>203</td>
<td>V</td>
<td>Vector Add Unsigned Halfword Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000240</td>
<td>V</td>
<td>205</td>
<td>V</td>
<td>Vector Add Unsigned Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000100</td>
<td>V</td>
<td>206</td>
<td>V</td>
<td>Vector Add Unsigned Quadword Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000080</td>
<td>V</td>
<td>204</td>
<td>V</td>
<td>Vector Add Unsigned Word Modulo</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000280</td>
<td>V</td>
<td>205</td>
<td>V</td>
<td>Vector Add Unsigned Word Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000040</td>
<td>V</td>
<td>238</td>
<td>V</td>
<td>Vector Logical AND</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000444</td>
<td>V</td>
<td>238</td>
<td>V</td>
<td>Vector Logical AND with Complement</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000502</td>
<td>V</td>
<td>226</td>
<td>V</td>
<td>Vector Average Signed Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000542</td>
<td>V</td>
<td>226</td>
<td>V</td>
<td>Vector Average Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000582</td>
<td>V</td>
<td>226</td>
<td>V</td>
<td>Vector Average Signed Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000422</td>
<td>V</td>
<td>227</td>
<td>V</td>
<td>Vector Average Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000442</td>
<td>V</td>
<td>227</td>
<td>V</td>
<td>Vector Average Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000482</td>
<td>V</td>
<td>227</td>
<td>V</td>
<td>Vector Average Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000054C</td>
<td>V</td>
<td>265</td>
<td>V</td>
<td>Vector Bit Permute Quadword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000054C</td>
<td>V</td>
<td>265</td>
<td>V</td>
<td>Vector Bit Permute Quadword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000034A</td>
<td>V</td>
<td>248</td>
<td>V</td>
<td>Vector Convert From Signed Fixed-Point Word To Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000030A</td>
<td>V</td>
<td>248</td>
<td>V</td>
<td>Vector Convert From Unsigned Fixed-Point Word To Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000702</td>
<td>V</td>
<td>263</td>
<td>V</td>
<td>Vector Count Leading Zeros Byte</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100007C2</td>
<td>V</td>
<td>263</td>
<td>V</td>
<td>Vector Count Leading Zeros Doubleword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000742</td>
<td>V</td>
<td>263</td>
<td>V</td>
<td>Vector Count Leading Zeros Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000782</td>
<td>V</td>
<td>263</td>
<td>V</td>
<td>Vector Count Leading Zeros Word</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100003C6</td>
<td>V</td>
<td>251</td>
<td>V</td>
<td>Vector Compare Bounds Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100000C6</td>
<td>V</td>
<td>252</td>
<td>V</td>
<td>Vector Compare Equal To Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000006</td>
<td>V</td>
<td>232</td>
<td>V</td>
<td>Vector Compare Equal To Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100000C7</td>
<td>V</td>
<td>233</td>
<td>V</td>
<td>Vector Compare Equal To Unsigned Doubleword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000046</td>
<td>V</td>
<td>233</td>
<td>V</td>
<td>Vector Compare Equal To Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000086</td>
<td>V</td>
<td>233</td>
<td>V</td>
<td>Vector Compare Equal To Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100001C6</td>
<td>V</td>
<td>252</td>
<td>V</td>
<td>Vector Compare Greater Than or Equal To Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100002C6</td>
<td>V</td>
<td>253</td>
<td>V</td>
<td>Vector Compare Greater Than Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000306</td>
<td>V</td>
<td>234</td>
<td>V</td>
<td>Vector Compare Greater Than Signed Byte</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100003C7</td>
<td>V</td>
<td>234</td>
<td>V</td>
<td>Vector Compare Greater Than Signed Doubleword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000346</td>
<td>V</td>
<td>234</td>
<td>V</td>
<td>Vector Compare Greater Than Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000386</td>
<td>V</td>
<td>235</td>
<td>V</td>
<td>Vector Compare Greater Than Signed Word</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000206</td>
<td>V</td>
<td>236</td>
<td>V</td>
<td>Vector Compare Greater Than Unsigned Byte</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100002C7</td>
<td>V</td>
<td>236</td>
<td>V</td>
<td>Vector Compare Greater Than Unsigned Doubleword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000246</td>
<td>V</td>
<td>236</td>
<td>V</td>
<td>Vector Compare Greater Than Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000286</td>
<td>V</td>
<td>237</td>
<td>V</td>
<td>Vector Compare Greater Than Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100003CA</td>
<td>V</td>
<td>247</td>
<td>V</td>
<td>Vector Convert From Single-Precision To Signed Fixed-Point Word Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000038A</td>
<td>V</td>
<td>247</td>
<td>V</td>
<td>Vector Convert From Single-Precision To Unsigned Fixed-Point Word Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000684</td>
<td>V</td>
<td>238</td>
<td>V</td>
<td>Vector Equivalence</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep. (^1)</td>
<td>Privilege (^1)</td>
<td>Page</td>
<td>Category (^1)</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>----------------</td>
<td>----------------</td>
<td>------</td>
<td>----------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000018A</td>
<td></td>
<td></td>
<td>254</td>
<td>V</td>
<td>vexptefp</td>
<td>Vector 2 Raised to the Exponent Estimate Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000050C</td>
<td></td>
<td></td>
<td>262</td>
<td>V</td>
<td>vgbbd</td>
<td>Vector Gather Bits by Byte by Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 100001CA</td>
<td></td>
<td></td>
<td>254</td>
<td>V</td>
<td>vlogefp</td>
<td>Vector Log Base 2 Estimate Single-Precision</td>
</tr>
<tr>
<td>VA</td>
<td>4 100002E</td>
<td></td>
<td></td>
<td>245</td>
<td>V</td>
<td>vmaddfp</td>
<td>Vector Multiply-Add Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000040A</td>
<td></td>
<td></td>
<td>246</td>
<td>V</td>
<td>vmaxfp</td>
<td>Vector Maximum Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000102</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxsb</td>
<td>Vector Maximum Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 100001C2</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxsd</td>
<td>Vector Maximum Signed Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000142</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxsh</td>
<td>Vector Maximum Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000182</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxsw</td>
<td>Vector Maximum Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000002</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxub</td>
<td>Vector Maximum Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 100000C2</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxud</td>
<td>Vector Maximum Unsigned Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000042</td>
<td></td>
<td></td>
<td>228</td>
<td>V</td>
<td>vmaxuh</td>
<td>Vector Maximum Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000082</td>
<td></td>
<td></td>
<td>229</td>
<td>V</td>
<td>vmaxuw</td>
<td>Vector Maximum Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000020</td>
<td></td>
<td></td>
<td>218</td>
<td>V</td>
<td>vmhaddshs</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA</td>
<td>4 1000021</td>
<td></td>
<td></td>
<td>218</td>
<td>V</td>
<td>vmhraddshs</td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000044A</td>
<td></td>
<td></td>
<td>246</td>
<td>V</td>
<td>vmminfp</td>
<td>Vector Minimum Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000302</td>
<td></td>
<td></td>
<td>230</td>
<td>V</td>
<td>vminsfb</td>
<td>Vector Minimum Signed Byte</td>
</tr>
<tr>
<td>X</td>
<td>4 100003C2</td>
<td></td>
<td></td>
<td>230</td>
<td>V</td>
<td>vminsdb</td>
<td>Vector Minimum Signed Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000342</td>
<td></td>
<td></td>
<td>230</td>
<td>V</td>
<td>vminsnh</td>
<td>Vector Minimum Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000382</td>
<td></td>
<td></td>
<td>231</td>
<td>V</td>
<td>vminsww</td>
<td>Vector Minimum Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000202</td>
<td></td>
<td></td>
<td>230</td>
<td>V</td>
<td>vminub</td>
<td>Vector Minimum Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 100002C2</td>
<td></td>
<td></td>
<td>230</td>
<td>V</td>
<td>vminud</td>
<td>Vector Minimum Unsigned Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000242</td>
<td></td>
<td></td>
<td>230</td>
<td>V</td>
<td>vminuh</td>
<td>Vector Minimum Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000282</td>
<td></td>
<td></td>
<td>231</td>
<td>V</td>
<td>vminuw</td>
<td>Vector Minimum Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000022</td>
<td></td>
<td></td>
<td>219</td>
<td>V</td>
<td>vmladduhm</td>
<td>Vector Multiply-Low-Add Signed Halfword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000000C</td>
<td></td>
<td></td>
<td>194</td>
<td>V</td>
<td>vmrghb</td>
<td>Vector Merge High Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000004C</td>
<td></td>
<td></td>
<td>194</td>
<td>V</td>
<td>vmrghh</td>
<td>Vector Merge High Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000008C</td>
<td></td>
<td></td>
<td>195</td>
<td>V</td>
<td>vmrghw</td>
<td>Vector Merge High Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000010C</td>
<td></td>
<td></td>
<td>194</td>
<td>V</td>
<td>vmrglb</td>
<td>Vector Merge Low Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000014C</td>
<td></td>
<td></td>
<td>194</td>
<td>V</td>
<td>vmrglh</td>
<td>Vector Merge Low Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000018C</td>
<td></td>
<td></td>
<td>195</td>
<td>V</td>
<td>vmrglw</td>
<td>Vector Merge Low Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000025</td>
<td></td>
<td></td>
<td>220</td>
<td>V</td>
<td>vmsummbm</td>
<td>Vector Multiply-Sum Mixed Byte Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000028</td>
<td></td>
<td></td>
<td>220</td>
<td>V</td>
<td>vmsumshm</td>
<td>Vector Multiply-Sum Signed Halfword Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000029</td>
<td></td>
<td></td>
<td>221</td>
<td>V</td>
<td>vmsumshs</td>
<td>Vector Multiply-Sum Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000024</td>
<td></td>
<td></td>
<td>219</td>
<td>V</td>
<td>vmsumubm</td>
<td>Vector Multiply-Sum Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000026</td>
<td></td>
<td></td>
<td>221</td>
<td>V</td>
<td>vmsumuhm</td>
<td>Vector Multiply-Sum Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000027</td>
<td></td>
<td></td>
<td>222</td>
<td>V</td>
<td>vmsumuhs</td>
<td>Vector Multiply-Sum Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000308</td>
<td></td>
<td></td>
<td>214</td>
<td>V</td>
<td>vmulesb</td>
<td>Vector Multiply Even Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000348</td>
<td></td>
<td></td>
<td>215</td>
<td>V</td>
<td>vmulesh</td>
<td>Vector Multiply Even Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000388</td>
<td></td>
<td></td>
<td>216</td>
<td>V</td>
<td>vmulesw</td>
<td>Vector Multiply Even Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000208</td>
<td></td>
<td></td>
<td>214</td>
<td>V</td>
<td>vmuleub</td>
<td>Vector Multiply Even Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000248</td>
<td></td>
<td></td>
<td>215</td>
<td>V</td>
<td>vmuleuh</td>
<td>Vector Multiply Even Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000288</td>
<td></td>
<td></td>
<td>216</td>
<td>V</td>
<td>vmuleuw</td>
<td>Vector Multiply Even Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000108</td>
<td></td>
<td></td>
<td>214</td>
<td>V</td>
<td>vmulosb</td>
<td>Vector Multiply Odd Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000148</td>
<td></td>
<td></td>
<td>215</td>
<td>V</td>
<td>vmulosh</td>
<td>Vector Multiply Odd Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000188</td>
<td></td>
<td></td>
<td>216</td>
<td>V</td>
<td>vmulosw</td>
<td>Vector Multiply Odd Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000008</td>
<td></td>
<td></td>
<td>214</td>
<td>V</td>
<td>vmuloub</td>
<td>Vector Multiply Odd Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000048</td>
<td></td>
<td></td>
<td>215</td>
<td>V</td>
<td>vmulouh</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000088</td>
<td>216</td>
<td>V</td>
<td>vmulouw</td>
<td>Vector Multiply Odd Unsigned Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000089</td>
<td>217</td>
<td>V</td>
<td>vmuluwm</td>
<td>Vector Multiply Unsigned Word Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000584</td>
<td>238</td>
<td>V</td>
<td>vnand</td>
<td>Vector NAND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>4 100002F</td>
<td>245</td>
<td>V</td>
<td>vnmsubfp</td>
<td>Vector Negative Multiply-Subtract Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000504</td>
<td>239</td>
<td>V</td>
<td>vnor</td>
<td>Vector Logical NOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000484</td>
<td>239</td>
<td>V</td>
<td>vor</td>
<td>Vector Logical OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>4 1000002B</td>
<td>198</td>
<td>V</td>
<td>vperm</td>
<td>Vector Permute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000030E</td>
<td>187</td>
<td>V</td>
<td>vpkpx</td>
<td>Vector Pack Pixel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100005CE</td>
<td>187</td>
<td>V</td>
<td>vpksdss</td>
<td>Vector Pack Signed Doubleword Signed Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100054E</td>
<td>188</td>
<td>V</td>
<td>vpksdus</td>
<td>Vector Pack Signed Doubleword Unsigned Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100018E</td>
<td>188</td>
<td>V</td>
<td>vpkswws</td>
<td>Vector Pack Signed Word Signed Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100014E</td>
<td>190</td>
<td>V</td>
<td>vpkswws</td>
<td>Vector Pack Signed Word Unsigned Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100044E</td>
<td>190</td>
<td>V</td>
<td>vpkudum</td>
<td>Vector Pack Unsized Doubleword Unsized Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10004CE</td>
<td>190</td>
<td>V</td>
<td>vpkudus</td>
<td>Vector Pack Unsized Doubleword Unsized Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100000E</td>
<td>190</td>
<td>V</td>
<td>vpkuhum</td>
<td>Vector Pack Unsized Halfword Unsized Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100008E</td>
<td>191</td>
<td>V</td>
<td>vpkhus</td>
<td>Vector Pack Unsized Halfword Unsigned Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100004E</td>
<td>191</td>
<td>V</td>
<td>vpkwum</td>
<td>Vector Pack Unsized Word Unsized Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000CE</td>
<td>191</td>
<td>V</td>
<td>vpkwus</td>
<td>Vector Pack Unsized Word Unsized Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000408</td>
<td>259</td>
<td>V</td>
<td>vpmmsmb</td>
<td>Vector Polynomial Multiply-Sum Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10004C8</td>
<td>259</td>
<td>V</td>
<td>vpmmsmd</td>
<td>Vector Polynomial Multiply-Sum Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000448</td>
<td>260</td>
<td>V</td>
<td>vpmmsmh</td>
<td>Vector Polynomial Multiply-Sum Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000488</td>
<td>260</td>
<td>V</td>
<td>vpmmsmw</td>
<td>Vector Polynomial Multiply-Sum Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000703</td>
<td>264</td>
<td>V</td>
<td>vpopcntb</td>
<td>Vector Population Count Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10007C3</td>
<td>264</td>
<td>V</td>
<td>vpopcntd</td>
<td>Vector Population Count Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000743</td>
<td>264</td>
<td>V</td>
<td>vpopcnth</td>
<td>Vector Population Count Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000783</td>
<td>264</td>
<td>V</td>
<td>vpopcntw</td>
<td>Vector Population Count Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100010A</td>
<td>255</td>
<td>V</td>
<td>vrefs</td>
<td>Vector Reciprocal Estimate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10002CA</td>
<td>250</td>
<td>V</td>
<td>vrfim</td>
<td>Vector Round to Single-Precision Integer toward -Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100020A</td>
<td>249</td>
<td>V</td>
<td>vrfin</td>
<td>Vector Round to Single-Precision Integer Nearest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100028A</td>
<td>249</td>
<td>V</td>
<td>vrfip</td>
<td>Vector Round to Single-Precision Integer toward +Infinity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100024A</td>
<td>249</td>
<td>V</td>
<td>vrfiz</td>
<td>Vector Round to Single-Precision Integer toward Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000004</td>
<td>240</td>
<td>V</td>
<td>vrlb</td>
<td>Vector Rotate Left Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000C4</td>
<td>240</td>
<td>V</td>
<td>vrlrd</td>
<td>Vector Rotate Left Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000044</td>
<td>240</td>
<td>V</td>
<td>vrlh</td>
<td>Vector Rotate Left Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000084</td>
<td>240</td>
<td>V</td>
<td>vrlw</td>
<td>Vector Rotate Left Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100014A</td>
<td>255</td>
<td>V</td>
<td>vrsqrtefp</td>
<td>Vector Reciprocal Square Root Estimate Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>4 100002A</td>
<td>199</td>
<td>V</td>
<td>vsel</td>
<td>Vector Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10001C4</td>
<td>200</td>
<td>V</td>
<td>vsll</td>
<td>Vector Shift Left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000104</td>
<td>241</td>
<td>V</td>
<td>vslb</td>
<td>Vector Shift Left Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100005C4</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vsld</td>
<td>Vector Shift Left Doubleword</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002C</td>
<td></td>
<td>200</td>
<td>V</td>
<td>vsldoi</td>
<td>Vector Shift Left Double by Octet Immediate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000144</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vslh</td>
<td>Vector Shift Left Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000040C</td>
<td></td>
<td>200</td>
<td>V</td>
<td>vslc</td>
<td>Vector Shift Left by Octet</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000184</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vslw</td>
<td>Vector Shift Left Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000020C</td>
<td></td>
<td>197</td>
<td>V</td>
<td>vspltb</td>
<td>Vector Splat Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000024C</td>
<td></td>
<td>197</td>
<td>V</td>
<td>vsploth</td>
<td>Vector Splat Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000030C</td>
<td></td>
<td>198</td>
<td>V</td>
<td>vspltisb</td>
<td>Vector Splat Immediate Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000034C</td>
<td></td>
<td>198</td>
<td>V</td>
<td>vspltish</td>
<td>Vector Splat Immediate Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000038C</td>
<td></td>
<td>198</td>
<td>V</td>
<td>vspltisw</td>
<td>Vector Splat Immediate Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000028C</td>
<td></td>
<td>197</td>
<td>V</td>
<td>vspltlw</td>
<td>Vector Splat Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100002C4</td>
<td></td>
<td>201</td>
<td>V</td>
<td>vsr</td>
<td>Vector Shift Right</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000304</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsrabo</td>
<td>Vector Shift Right Algebraic Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100003C4</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsrad</td>
<td>Vector Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000344</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsrah</td>
<td>Vector Shift Right Algebraic Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000384</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsraw</td>
<td>Vector Shift Right Algebraic Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000204</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrb</td>
<td>Vector Shift Right Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100006C4</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrd</td>
<td>Vector Shift Right Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000244</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrh</td>
<td>Vector Shift Right Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000044C</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vsro</td>
<td>Vector Shift Right by Octet</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000284</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrw</td>
<td>Vector Shift Right Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000540</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsbcuq</td>
<td>Vector Subtract &amp; write Carry Uns signed Quadword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000580</td>
<td></td>
<td>208</td>
<td>V</td>
<td>vsbcuw</td>
<td>Vector Subtract and Write Carry-Out Uns signed Word</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000003F</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsubecuq</td>
<td>Vector Subtract Extended &amp; write Carry Uns signed Quadword</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000003E</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsubeuqm</td>
<td>Vector Subtract Extended Uns signed Quadword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100004A</td>
<td></td>
<td>244</td>
<td>V</td>
<td>vsubfp</td>
<td>Vector Subtract Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000700</td>
<td></td>
<td>208</td>
<td>V</td>
<td>vsubbs</td>
<td>Vector Subtract Signed Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000740</td>
<td></td>
<td>208</td>
<td>V</td>
<td>vsubshs</td>
<td>Vector Subtract Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000780</td>
<td></td>
<td>209</td>
<td>V</td>
<td>vsubsws</td>
<td>Vector Subtract Signed Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000400</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsububm</td>
<td>Vector Subtract Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000600</td>
<td></td>
<td>211</td>
<td>V</td>
<td>vsububs</td>
<td>Vector Subtract Unsigned Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100004C0</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsbudm</td>
<td>Vector Subtract Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000440</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsubuhm</td>
<td>Vector Subtract Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000640</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsubuh</td>
<td>Vector Subtract Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000500</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsbucm</td>
<td>Vector Subtract Unsigned Quadsword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000480</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsbuwum</td>
<td>Vector Subtract Unsigned Word Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000680</td>
<td></td>
<td>211</td>
<td>V</td>
<td>vsbuxws</td>
<td>Vector Subtract Unsigned Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000688</td>
<td></td>
<td>223</td>
<td>V</td>
<td>vsum2sws</td>
<td>Vector Sum across Half Signed Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000708</td>
<td></td>
<td>224</td>
<td>V</td>
<td>vsum4sbs</td>
<td>Vector Sum across Quarter Signed Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000648</td>
<td></td>
<td>224</td>
<td>V</td>
<td>vsum4shs</td>
<td>Vector Sum across Quarter Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000608</td>
<td></td>
<td>225</td>
<td>V</td>
<td>vsum4ubs</td>
<td>Vector Sum across Quarter Uns signed Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000788</td>
<td></td>
<td>223</td>
<td>V</td>
<td>vsumsws</td>
<td>Vector Sum across Signed Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000034E</td>
<td></td>
<td>190</td>
<td>V</td>
<td>vupkhpx</td>
<td>Vector Unpack High Pixel</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000020E</td>
<td></td>
<td>193</td>
<td>V</td>
<td>vupkhsb</td>
<td>Vector Unpack High Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000024E</td>
<td></td>
<td>193</td>
<td>V</td>
<td>vupkhsb</td>
<td>Vector Unpack High Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000064E</td>
<td></td>
<td>193</td>
<td>V</td>
<td>vupkhsw</td>
<td>Vector Unpack High Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100003CE</td>
<td></td>
<td>192</td>
<td>V</td>
<td>vupklpx</td>
<td>Vector Unpack Low Pixel</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000028E</td>
<td></td>
<td>193</td>
<td>V</td>
<td>vupklspb</td>
<td>Vector Unpack Low Signed Byte</td>
</tr>
<tr>
<td>Primary</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100002CE</td>
<td></td>
<td></td>
<td>193</td>
<td>V</td>
<td>vupklsh Vector Unpack Low Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100006CE</td>
<td></td>
<td></td>
<td>193</td>
<td>V</td>
<td>vupklsw Vector Unpack Low Signed Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100004C4</td>
<td></td>
<td></td>
<td>239</td>
<td>V</td>
<td>vxor Vector Logical XOR</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000508</td>
<td></td>
<td></td>
<td>256</td>
<td>V</td>
<td>vcipher Vector AES Cipher</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000509</td>
<td></td>
<td></td>
<td>256</td>
<td>V</td>
<td>vcipherlast Vector AES Cipher Last</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000548</td>
<td></td>
<td></td>
<td>257</td>
<td>V</td>
<td>vncipher Vector AES Inverse Cipher</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000549</td>
<td></td>
<td></td>
<td>257</td>
<td>V</td>
<td>vncipherlast Vector AES Inverse Cipher Last</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100005C8</td>
<td></td>
<td></td>
<td>257</td>
<td>V</td>
<td>vsbox Vector AES S-Box</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100006C2</td>
<td></td>
<td></td>
<td>258</td>
<td>V</td>
<td>vshasigmad Vector SHA-512 Sigma Doubleword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000682</td>
<td></td>
<td></td>
<td>258</td>
<td>V</td>
<td>vshasigmaw Vector SHA-256 Sigma Word</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>4 100002D</td>
<td></td>
<td></td>
<td>261</td>
<td>V</td>
<td>vpermxor Vector Permute and Exclusive-OR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC0007BC</td>
<td></td>
<td></td>
<td>143</td>
<td>V</td>
<td>fpermxw Floating Merge Even Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC0006BC</td>
<td></td>
<td></td>
<td>143</td>
<td>V</td>
<td>fpermow Floating Merge Odd Word</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000498</td>
<td></td>
<td></td>
<td>393</td>
<td>V</td>
<td>lxsdx Load VSR Scalar Doubleword Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000098</td>
<td></td>
<td></td>
<td>393</td>
<td>V</td>
<td>lxsiwax Load VSR Scalar as Integer Word Algebraic Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C0001A8</td>
<td></td>
<td></td>
<td>394</td>
<td>V</td>
<td>lxsiwzx Load VSR Scalar as Integer Word and Zero Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000418</td>
<td></td>
<td></td>
<td>394</td>
<td>V</td>
<td>lxsspx Load VSR Scalar Single-Precision Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000698</td>
<td></td>
<td></td>
<td>395</td>
<td>V</td>
<td>lxvd2x Load VSR Vector Doubleword*2 Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000298</td>
<td></td>
<td></td>
<td>395</td>
<td>V</td>
<td>lxvdxs Load VSR Vector Doubleword &amp; Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000618</td>
<td></td>
<td></td>
<td>396</td>
<td>V</td>
<td>lxw4x Load VSR Vector Word*4 Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000066</td>
<td></td>
<td></td>
<td>109</td>
<td>V</td>
<td>mfvsrd Move From VSR Doubleword</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C0000E6</td>
<td></td>
<td></td>
<td>109</td>
<td>V</td>
<td>mfvsrwz Move From VSR Word and Zero</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C000166</td>
<td></td>
<td></td>
<td>110</td>
<td>V</td>
<td>mtvsrd Move To VSR Doubleword</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C0001A6</td>
<td></td>
<td></td>
<td>110</td>
<td>V</td>
<td>mtvsrwa Move To VSR Word Algebraic</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C0001E6</td>
<td></td>
<td></td>
<td>111</td>
<td>V</td>
<td>mtvsrwz Move To VSR Word and Zero</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000598</td>
<td></td>
<td></td>
<td>396</td>
<td>V</td>
<td>stxsdx Store VSR Scalar Doubleword Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000118</td>
<td></td>
<td></td>
<td>394</td>
<td>V</td>
<td>stxsiwx Store VSR Scalar as Integer Word Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000518</td>
<td></td>
<td></td>
<td>394</td>
<td>V</td>
<td>stxsspx Store VSR Scalar Single-Precision Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000798</td>
<td></td>
<td></td>
<td>398</td>
<td>V</td>
<td>stvd2x Store VSR Vector Doubleword*2 Indexed</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000718</td>
<td></td>
<td></td>
<td>398</td>
<td>V</td>
<td>stxxw4x Store VSR Vector Word*4 Indexed</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100007BC</td>
<td></td>
<td></td>
<td>196</td>
<td>V</td>
<td>vmrgew Vector Merge Even Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100006BC</td>
<td></td>
<td></td>
<td>196</td>
<td>V</td>
<td>vmrgow Vector Merge Odd Word</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000564</td>
<td></td>
<td></td>
<td>399</td>
<td>V</td>
<td>xsabdp VSX Scalar Absolute Value Double-Precision</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00001100</td>
<td></td>
<td></td>
<td>400</td>
<td>V</td>
<td>xsadddp VSX Scalar Add Double-Precision</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000000</td>
<td></td>
<td></td>
<td>405</td>
<td>V</td>
<td>xsaddsp VSX Scalar Add Single-Precision</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000158</td>
<td></td>
<td></td>
<td>407</td>
<td>V</td>
<td>xscmpodp VSX Scalar Compare Ordered Double-Precision</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000118</td>
<td></td>
<td></td>
<td>409</td>
<td>V</td>
<td>xscmpudp VSX Scalar Compare Unordered Double-Precision</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000580</td>
<td></td>
<td></td>
<td>411</td>
<td>V</td>
<td>xscpshdp VSX Scalar Copy Sign Double-Precision</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000424</td>
<td></td>
<td></td>
<td>412</td>
<td>V</td>
<td>xsccdpdp VSX Scalar Convert Double-Precision to Single-Precision</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000042C</td>
<td></td>
<td></td>
<td>413</td>
<td>V</td>
<td>xsccdpdpsn VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000560</td>
<td></td>
<td></td>
<td>422</td>
<td>V</td>
<td>xsccdpdpsdxs VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000160</td>
<td></td>
<td></td>
<td>413</td>
<td>V</td>
<td>xsccdpdpsxws VSX Scalar Convert Double-Precision to Signed Fixed-Point Word Saturate</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000520</td>
<td></td>
<td></td>
<td>416</td>
<td>V</td>
<td>xsccdpuxds VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000120</td>
<td></td>
<td></td>
<td>418</td>
<td>V</td>
<td>xsccdpuxws VSX Scalar Convert Double-Precision to Signed Fixed-Point Word Saturate</td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000524</td>
<td></td>
<td></td>
<td>420</td>
<td>V</td>
<td>xsccspdp VSX Scalar Convert Single-Precision to Double-Precision (p=1)</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>0000052C</td>
<td></td>
<td>422</td>
<td>VSX</td>
<td>xscvspdpn</td>
<td>Scalar Convert Single-Precision to Double-Precision format Non-signalling</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>000005E0</td>
<td></td>
<td>423</td>
<td>VSX</td>
<td>xscvsxddp</td>
<td>VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>000004E0</td>
<td></td>
<td>423</td>
<td>VSX</td>
<td>xscvsxsp</td>
<td>VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>000005A0</td>
<td></td>
<td>424</td>
<td>VSX</td>
<td>xscvxuddp</td>
<td>VSX Scalar Convert Unsigned Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>000004A0</td>
<td></td>
<td>424</td>
<td>VSX</td>
<td>xscvxsp</td>
<td>VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>000001C0</td>
<td></td>
<td>425</td>
<td>VSX</td>
<td>xsdivdp</td>
<td>VSX Scalar Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>000000C0</td>
<td></td>
<td>427</td>
<td>VSX</td>
<td>xsdivsp</td>
<td>VSX Scalar Divide Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000108</td>
<td></td>
<td>429</td>
<td>VSX</td>
<td>xsmaddadp</td>
<td>VSX Scalar Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000008</td>
<td></td>
<td>432</td>
<td>VSX</td>
<td>xsmaddasp</td>
<td>VSX Scalar Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000148</td>
<td></td>
<td>429</td>
<td>VSX</td>
<td>xsmaddmdp</td>
<td>VSX Scalar Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000048</td>
<td></td>
<td>432</td>
<td>VSX</td>
<td>xsmaddmsp</td>
<td>VSX Scalar Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000500</td>
<td></td>
<td>435</td>
<td>VSX</td>
<td>xsmaxdp</td>
<td>VSX Scalar Maximum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000540</td>
<td></td>
<td>437</td>
<td>VSX</td>
<td>xsmindp</td>
<td>VSX Scalar Minimum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000188</td>
<td></td>
<td>439</td>
<td>VSX</td>
<td>xmsubadp</td>
<td>VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000088</td>
<td></td>
<td>442</td>
<td>VSX</td>
<td>xmsubasp</td>
<td>VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00001C8</td>
<td></td>
<td>439</td>
<td>VSX</td>
<td>xmsubmdp</td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000C8</td>
<td></td>
<td>442</td>
<td>VSX</td>
<td>xmsubmsp</td>
<td>VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000180</td>
<td></td>
<td>445</td>
<td>VSX</td>
<td>xsmuldp</td>
<td>VSX Scalar Multiply Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>0000080</td>
<td></td>
<td>447</td>
<td>VSX</td>
<td>xsmulsp</td>
<td>VSX Scalar Multiply Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>000005A4</td>
<td></td>
<td>449</td>
<td>VSX</td>
<td>xsnabsp</td>
<td>VSX Scalar Negative Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>000005E4</td>
<td></td>
<td>449</td>
<td>VSX</td>
<td>xsnegdp</td>
<td>VSX Scalar Negate Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000508</td>
<td></td>
<td>450</td>
<td>VSX</td>
<td>xsnmaddadp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000408</td>
<td></td>
<td>455</td>
<td>VSX</td>
<td>xsnmaddasp</td>
<td>VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000548</td>
<td></td>
<td>450</td>
<td>VSX</td>
<td>xsnmaddmdp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000448</td>
<td></td>
<td>455</td>
<td>VSX</td>
<td>xsnmaddmsp</td>
<td>VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000588</td>
<td></td>
<td>458</td>
<td>VSX</td>
<td>xsnsubadp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>00000488</td>
<td></td>
<td>461</td>
<td>VSX</td>
<td>xsnsubasp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>000005C8</td>
<td></td>
<td>458</td>
<td>VSX</td>
<td>xsnsubmdp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>000004C8</td>
<td></td>
<td>461</td>
<td>VSX</td>
<td>xsnsubmsp</td>
<td>VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>0000124</td>
<td></td>
<td>464</td>
<td>VSX</td>
<td>xsrdpi</td>
<td>VSX Scalar Round to Double-Precision Integer</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>00001AC</td>
<td></td>
<td>465</td>
<td>VSX</td>
<td>xsrdpic</td>
<td>VSX Scalar Round to Double-Precision Integer using Current rounding mode</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>00001E4</td>
<td></td>
<td>466</td>
<td>VSX</td>
<td>xsrdpim</td>
<td>VSX Scalar Round to Double-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00001A4</td>
<td></td>
<td></td>
<td>466</td>
<td>VSX</td>
<td>xsrdpip</td>
<td>VSX Scalar Round to Double-Precision Integer toward +Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000164</td>
<td></td>
<td></td>
<td>467</td>
<td>VSX</td>
<td>xsrdpiz</td>
<td>VSX Scalar Round to Double-Precision Integer toward Zero</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000168</td>
<td></td>
<td></td>
<td>468</td>
<td>VSX</td>
<td>xsredp</td>
<td>VSX Scalar Reciprocal Estimate Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000668</td>
<td></td>
<td></td>
<td>469</td>
<td>VSX</td>
<td>xsresp</td>
<td>VSX Scalar Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000464</td>
<td></td>
<td></td>
<td>470</td>
<td>VSX</td>
<td>xsrsp</td>
<td>VSX Scalar Round to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000128</td>
<td></td>
<td></td>
<td>471</td>
<td>VSX</td>
<td>xsrsqrtedp</td>
<td>VSX Scalar Reciprocal Square Root Estimate Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000228</td>
<td></td>
<td></td>
<td>472</td>
<td>VSX</td>
<td>xsrsqrtesp</td>
<td>VSX Scalar Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000012C</td>
<td></td>
<td></td>
<td>473</td>
<td>VSX</td>
<td>xssqrtdp</td>
<td>VSX Scalar Square Root Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000002C</td>
<td></td>
<td></td>
<td>474</td>
<td>VSX</td>
<td>xssqrtsp</td>
<td>VSX Scalar Square Root Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000140</td>
<td></td>
<td></td>
<td>475</td>
<td>VSX</td>
<td>xssubdp</td>
<td>VSX Scalar Subtract Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000040</td>
<td></td>
<td></td>
<td>477</td>
<td>VSX</td>
<td>xssubsp</td>
<td>VSX Scalar Subtract Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00001E8</td>
<td></td>
<td></td>
<td>479</td>
<td>VSX</td>
<td>xstdivdp</td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00001A8</td>
<td></td>
<td></td>
<td>480</td>
<td>VSX</td>
<td>xstsqrdp</td>
<td>VSX Scalar Test for software Square Root Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000764</td>
<td></td>
<td></td>
<td>480</td>
<td>VSX</td>
<td>xvabsdp</td>
<td>VSX Vector Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000664</td>
<td></td>
<td></td>
<td>481</td>
<td>VSX</td>
<td>xvabssp</td>
<td>VSX Vector Absolute Value Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000300</td>
<td></td>
<td></td>
<td>482</td>
<td>VSX</td>
<td>xvadddp</td>
<td>VSX Vector Add Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000200</td>
<td></td>
<td></td>
<td>486</td>
<td>VSX</td>
<td>xvaddsp</td>
<td>VSX Vector Add Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000318</td>
<td></td>
<td></td>
<td>488</td>
<td>VSX</td>
<td>xvcmpeqdp</td>
<td>VSX Vector Compare Equal To Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000718</td>
<td></td>
<td></td>
<td>488</td>
<td>VSX</td>
<td>xvcmpeqdp.</td>
<td>VSX Vector Compare Equal To Double-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000218</td>
<td></td>
<td></td>
<td>489</td>
<td>VSX</td>
<td>xvcmpeqsp</td>
<td>VSX Vector Compare Equal To Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000618</td>
<td></td>
<td></td>
<td>489</td>
<td>VSX</td>
<td>xvcmpeqsp.</td>
<td>VSX Vector Compare Equal To Single-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000398</td>
<td></td>
<td></td>
<td>490</td>
<td>VSX</td>
<td>xvcmpgedp</td>
<td>VSX Vector Compare Greater Than or Equal To Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000798</td>
<td></td>
<td></td>
<td>490</td>
<td>VSX</td>
<td>xvcmpgedp.</td>
<td>VSX Vector Compare Greater Than or Equal To Double-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000298</td>
<td></td>
<td></td>
<td>491</td>
<td>VSX</td>
<td>xvcmpgesp</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000698</td>
<td></td>
<td></td>
<td>491</td>
<td>VSX</td>
<td>xvcmpgesp.</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000358</td>
<td></td>
<td></td>
<td>492</td>
<td>VSX</td>
<td>xvcmpgtdp</td>
<td>VSX Vector Compare Greater Than Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000758</td>
<td></td>
<td></td>
<td>492</td>
<td>VSX</td>
<td>xvcmpgtdp.</td>
<td>VSX Vector Compare Greater Than Double-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000258</td>
<td></td>
<td></td>
<td>493</td>
<td>VSX</td>
<td>xvcmpgtpsp</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000658</td>
<td></td>
<td></td>
<td>493</td>
<td>VSX</td>
<td>xvcmpgtpsp.</td>
<td>VSX Vector Compare Greater Than Single-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000780</td>
<td></td>
<td></td>
<td>494</td>
<td>VSX</td>
<td>xvcpsgndp</td>
<td>VSX Vector Copy Sign Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000680</td>
<td></td>
<td></td>
<td>494</td>
<td>VSX</td>
<td>xvcpsgnsp</td>
<td>VSX Vector Copy Sign Single-Precision</td>
</tr>
<tr>
<td>X2</td>
<td>60 F0000624</td>
<td></td>
<td></td>
<td>495</td>
<td>VSX</td>
<td>xvcvdpps</td>
<td>VSX Vector Convert Double-Precision to Single-Precision</td>
</tr>
<tr>
<td>X2</td>
<td>60 F0000760</td>
<td></td>
<td></td>
<td>496</td>
<td>VSX</td>
<td>xvcvdpsxds</td>
<td>VSX Vector Convert Double-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>X2</td>
<td>60 F0000360</td>
<td></td>
<td></td>
<td>498</td>
<td>VSX</td>
<td>xvcvdpsxws</td>
<td>VSX Vector Convert Double-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000720</td>
<td></td>
<td></td>
<td>500</td>
<td>VSX</td>
<td>xcvdpuxds</td>
<td>VSX Vector Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000320</td>
<td></td>
<td></td>
<td>502</td>
<td>VSX</td>
<td>xcvdpuxws</td>
<td>VSX Vector Convert Double-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000724</td>
<td></td>
<td></td>
<td>504</td>
<td>VSX</td>
<td>xcvspdp</td>
<td>VSX Vector Convert Single-Precision to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000660</td>
<td></td>
<td></td>
<td>505</td>
<td>VSX</td>
<td>xcvspxsd</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000260</td>
<td></td>
<td></td>
<td>507</td>
<td>VSX</td>
<td>xcvspxsxws</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000620</td>
<td></td>
<td></td>
<td>509</td>
<td>VSX</td>
<td>xcvspuxds</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000220</td>
<td></td>
<td></td>
<td>511</td>
<td>VSX</td>
<td>xcvspuxws</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007E0</td>
<td></td>
<td></td>
<td>513</td>
<td>VSX</td>
<td>xvcvsxddp</td>
<td>VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006E0</td>
<td></td>
<td></td>
<td>513</td>
<td>VSX</td>
<td>xvcvsxsp</td>
<td>VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003E0</td>
<td></td>
<td></td>
<td>514</td>
<td>VSX</td>
<td>xvcvsxwp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00002E0</td>
<td></td>
<td></td>
<td>514</td>
<td>VSX</td>
<td>xvcvsxwp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007A0</td>
<td></td>
<td></td>
<td>515</td>
<td>VSX</td>
<td>xvcvuxdp</td>
<td>VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006A0</td>
<td></td>
<td></td>
<td>515</td>
<td>VSX</td>
<td>xvcvuxdp</td>
<td>VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003A0</td>
<td></td>
<td></td>
<td>516</td>
<td>VSX</td>
<td>xvcvuxwp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00002A0</td>
<td></td>
<td></td>
<td>516</td>
<td>VSX</td>
<td>xvcvuxwp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003C0</td>
<td></td>
<td></td>
<td>517</td>
<td>VSX</td>
<td>xvdvdp</td>
<td>VSX Vector Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002C0</td>
<td></td>
<td></td>
<td>519</td>
<td>VSX</td>
<td>xdivsp</td>
<td>VSX Vector Divide Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000308</td>
<td></td>
<td></td>
<td>521</td>
<td>VSX</td>
<td>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000208</td>
<td></td>
<td></td>
<td>521</td>
<td>VSX</td>
<td>xvmaddasp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000348</td>
<td></td>
<td></td>
<td>524</td>
<td>VSX</td>
<td>xvmaddmdp</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000248</td>
<td></td>
<td></td>
<td>524</td>
<td>VSX</td>
<td>xvmaddmsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000700</td>
<td></td>
<td></td>
<td>527</td>
<td>VSX</td>
<td>xvmaddp</td>
<td>VSX Vector Maximum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000600</td>
<td></td>
<td></td>
<td>529</td>
<td>VSX</td>
<td>xvmaddsp</td>
<td>VSX Vector Maximum Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000740</td>
<td></td>
<td></td>
<td>531</td>
<td>VSX</td>
<td>xvmindp</td>
<td>VSX Vector Minimum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000640</td>
<td></td>
<td></td>
<td>533</td>
<td>VSX</td>
<td>xvmindsp</td>
<td>VSX Vector Minimum Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000388</td>
<td></td>
<td></td>
<td>535</td>
<td>VSX</td>
<td>xvmsubadp</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000288</td>
<td></td>
<td></td>
<td>535</td>
<td>VSX</td>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003C8</td>
<td></td>
<td></td>
<td>538</td>
<td>VSX</td>
<td>xvmsubmdp</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002C8</td>
<td></td>
<td></td>
<td>538</td>
<td>VSX</td>
<td>xvmsubmsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000380</td>
<td></td>
<td></td>
<td>541</td>
<td>VSX</td>
<td>xvmulpd</td>
<td>VSX Vector Multiply Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000280</td>
<td></td>
<td></td>
<td>543</td>
<td>VSX</td>
<td>xvmulsp</td>
<td>VSX Vector Multiply Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007A4</td>
<td></td>
<td></td>
<td>545</td>
<td>VSX</td>
<td>xvnabsdp</td>
<td>VSX Vector Negative Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006A4</td>
<td></td>
<td></td>
<td>545</td>
<td>VSX</td>
<td>xvnabssp</td>
<td>VSX Vector Negative Absolute Value Single-Precision</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007E4</td>
<td></td>
<td></td>
<td>546</td>
<td>VSX</td>
<td>xvnegdp</td>
<td>VSX Vector Negate Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006E4</td>
<td></td>
<td></td>
<td>546</td>
<td>VSX</td>
<td>xvnegsp</td>
<td>VSX Vector Negate Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000708</td>
<td></td>
<td></td>
<td>547</td>
<td>VSX</td>
<td>xvnaddadp</td>
<td>VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000508</td>
<td></td>
<td></td>
<td>547</td>
<td>VSX</td>
<td>xvnaddasp</td>
<td>VSX Vector Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000748</td>
<td></td>
<td></td>
<td>552</td>
<td>VSX</td>
<td>xvnaddmdp</td>
<td>VSX Vector Negative Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000648</td>
<td></td>
<td></td>
<td>552</td>
<td>VSX</td>
<td>xvnaddmsp</td>
<td>VSX Vector Negative Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000788</td>
<td></td>
<td></td>
<td>555</td>
<td>VSX</td>
<td>xvnmsubadp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000688</td>
<td></td>
<td></td>
<td>555</td>
<td>VSX</td>
<td>xvnmsubasp</td>
<td>VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00007C8</td>
<td></td>
<td></td>
<td>558</td>
<td>VSX</td>
<td>xvnmsubmdp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00006C8</td>
<td></td>
<td></td>
<td>558</td>
<td>VSX</td>
<td>xvnmsubmsp</td>
<td>VSX Vector Negative Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000324</td>
<td></td>
<td></td>
<td>561</td>
<td>VSX</td>
<td>xvrdpi</td>
<td>VSX Vector Round to Double-Precision Integer</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003AC</td>
<td></td>
<td></td>
<td>561</td>
<td>VSX</td>
<td>xvrdpic</td>
<td>VSX Vector Round to Double-Precision Integer using Current rounding mode</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003E4</td>
<td></td>
<td></td>
<td>562</td>
<td>VSX</td>
<td>xvrdpim</td>
<td>VSX Vector Round to Double-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003A4</td>
<td></td>
<td></td>
<td>562</td>
<td>VSX</td>
<td>xvrdpip</td>
<td>VSX Vector Round to Double-Precision Integer toward +Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003E4</td>
<td></td>
<td></td>
<td>563</td>
<td>VSX</td>
<td>xvrdpiz</td>
<td>VSX Vector Round to Double-Precision Integer toward Zero</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000368</td>
<td></td>
<td></td>
<td>564</td>
<td>VSX</td>
<td>xvrredp</td>
<td>VSX Vector Reciprocal Estimate Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000268</td>
<td></td>
<td></td>
<td>565</td>
<td>VSX</td>
<td>xvrresp</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000224</td>
<td></td>
<td></td>
<td>566</td>
<td>VSX</td>
<td>xvrspi</td>
<td>VSX Vector Round to Single-Precision Integer</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00002AC</td>
<td></td>
<td></td>
<td>566</td>
<td>VSX</td>
<td>xvrspic</td>
<td>VSX Vector Round to Single-Precision Integer using Current rounding mode</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00002E4</td>
<td></td>
<td></td>
<td>567</td>
<td>VSX</td>
<td>xvrspim</td>
<td>VSX Vector Round to Single-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00002A4</td>
<td></td>
<td></td>
<td>567</td>
<td>VSX</td>
<td>xvrspip</td>
<td>VSX Vector Round to Single-Precision Integer toward +Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000264</td>
<td></td>
<td></td>
<td>568</td>
<td>VSX</td>
<td>xvrspiz</td>
<td>VSX Vector Round to Single-Precision Integer toward Zero</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000328</td>
<td></td>
<td></td>
<td>568</td>
<td>VSX</td>
<td>xvrsqrtedp</td>
<td>VSX Vector Reciprocal Square Root Estimate Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000228</td>
<td></td>
<td></td>
<td>570</td>
<td>VSX</td>
<td>xvrsqrtesp</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000032C</td>
<td></td>
<td></td>
<td>571</td>
<td>VSX</td>
<td>xvsqrtadp</td>
<td>VSX Vector Square Root Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000022C</td>
<td></td>
<td></td>
<td>572</td>
<td>VSX</td>
<td>xvsqrtsp</td>
<td>VSX Vector Square Root Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000340</td>
<td></td>
<td></td>
<td>573</td>
<td>VSX</td>
<td>xvsubdp</td>
<td>VSX Vector Subtract Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000240</td>
<td></td>
<td></td>
<td>575</td>
<td>VSX</td>
<td>xsvsubsp</td>
<td>VSX Vector Subtract Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003E8</td>
<td></td>
<td></td>
<td>577</td>
<td>VSX</td>
<td>xvtdivdp</td>
<td>VSX Vector Test for software Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002E8</td>
<td></td>
<td></td>
<td>578</td>
<td>VSX</td>
<td>xvtdivsp</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003A8</td>
<td></td>
<td></td>
<td>579</td>
<td>VSX</td>
<td>xvt.sqrtadp</td>
<td>VSX Vector Test for software Square Root Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00002A8</td>
<td></td>
<td></td>
<td>579</td>
<td>VSX</td>
<td>xvt.sqrtsp</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000410</td>
<td></td>
<td></td>
<td>580</td>
<td>VSX</td>
<td>xxland</td>
<td>VSX Logical AND</td>
</tr>
</tbody>
</table>
### Appendix G. Power ISA Instruction Set Sorted by Category

**Table**

<table>
<thead>
<tr>
<th>Format</th>
<th>Primary Base (hex)</th>
<th>Opcode</th>
<th>Mode Dep.1</th>
<th>Privilege1</th>
<th>Page</th>
<th>Category1</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX3</td>
<td>60 00000450</td>
<td>580</td>
<td>VSX</td>
<td>xlandc</td>
<td></td>
<td>VSX Logical AND with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 0000050D</td>
<td>581</td>
<td>VSX</td>
<td>xxleqv</td>
<td></td>
<td>VSX Logical Equivalence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000590</td>
<td>581</td>
<td>VSX</td>
<td>xxinand</td>
<td></td>
<td>VSX Logical NAND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000510</td>
<td>582</td>
<td>VSX</td>
<td>xxinor</td>
<td></td>
<td>VSX Logical NOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000490</td>
<td>582</td>
<td>VSX</td>
<td>xxlor</td>
<td></td>
<td>VSX Logical OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000550</td>
<td>582</td>
<td>VSX</td>
<td>xxlorc</td>
<td></td>
<td>VSX Logical OR with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 000004D0</td>
<td>583</td>
<td>VSX</td>
<td>xxixor</td>
<td></td>
<td>VSX Logical XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000090</td>
<td>584</td>
<td>VSX</td>
<td>xxmrghw</td>
<td></td>
<td>VSX Merge High Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000190</td>
<td>584</td>
<td>VSX</td>
<td>xxmrglw</td>
<td></td>
<td>VSX Merge Low Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000050</td>
<td>585</td>
<td>VSX</td>
<td>xxpermdl</td>
<td></td>
<td>VSX Permute Doubleword Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX4</td>
<td>60 00000030</td>
<td>585</td>
<td>VSX</td>
<td>xxsel</td>
<td></td>
<td>VSX Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000010</td>
<td>586</td>
<td>VSX</td>
<td>xxslidwi</td>
<td></td>
<td>VSX Shift Left Double by Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60 00000290</td>
<td>586</td>
<td>VSX</td>
<td>xxsplitw</td>
<td></td>
<td>VSX Splat Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00007C</td>
<td>791</td>
<td>WT</td>
<td>wait</td>
<td></td>
<td>Wait for Interrupt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.
Appendix H. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, sorted by opcode.

<table>
<thead>
<tr>
<th>Format</th>
<th>Opcode</th>
<th>Mode Dep.</th>
<th>Privilege</th>
<th>Page</th>
<th>Category</th>
<th>Mnemonic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>D 2</td>
<td>080000000</td>
<td>D</td>
<td>81</td>
<td>64</td>
<td>tdi</td>
<td>Trap Doubleword Immediate</td>
</tr>
<tr>
<td>D 3</td>
<td>000000000</td>
<td>D</td>
<td>80</td>
<td>B</td>
<td>twi</td>
<td>Trap Word Immediate</td>
</tr>
<tr>
<td>VX 4</td>
<td>100000000</td>
<td>V</td>
<td>203</td>
<td>V</td>
<td>vaddubm</td>
<td>Vector Add Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VX 4</td>
<td>100000002</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxub</td>
<td>Vector Maximum Unsigned Byte</td>
</tr>
<tr>
<td>VX 4</td>
<td>100000004</td>
<td>V</td>
<td>240</td>
<td>V</td>
<td>vrlb</td>
<td>Vector Rotate Left Byte</td>
</tr>
<tr>
<td>VC 4</td>
<td>100000006</td>
<td>V</td>
<td>232</td>
<td>V</td>
<td>vcmpequb</td>
<td>Vector Compare Equal To Unsigned Byte</td>
</tr>
<tr>
<td>VX 4</td>
<td>100000008</td>
<td>V</td>
<td>214</td>
<td>V</td>
<td>vmuloub</td>
<td>Vector Multiply Odd Unsigned Byte</td>
</tr>
<tr>
<td>X 4</td>
<td>10000010</td>
<td>LMA</td>
<td>681</td>
<td>LMA</td>
<td>mulhhuw</td>
<td>Multiply High Halfword to Word Unsigned</td>
</tr>
<tr>
<td>XO 4</td>
<td>10000018</td>
<td>LMA</td>
<td>678</td>
<td>LMA</td>
<td>machhwu</td>
<td>Multiply Accumulate High Halfword to Word Modulo Unsigned</td>
</tr>
<tr>
<td>XO 4</td>
<td>10000018</td>
<td>LMA</td>
<td>678</td>
<td>LMA</td>
<td>machhwuo</td>
<td>Multiply Accumulate High Halfword to Word Modulo Unsized &amp; record OV</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000020</td>
<td>V</td>
<td>218</td>
<td>V</td>
<td>vmhaddshs</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000021</td>
<td>V</td>
<td>218</td>
<td>V</td>
<td>vmhraddshs</td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000022</td>
<td>V</td>
<td>219</td>
<td>V</td>
<td>vmladduhm</td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000024</td>
<td>V</td>
<td>219</td>
<td>V</td>
<td>vmsumumhm</td>
<td>Vector Multiply-Sum Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000025</td>
<td>V</td>
<td>220</td>
<td>V</td>
<td>vmsummbm</td>
<td>Vector Multiply-Sum Mixed Byte Modulo</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000026</td>
<td>V</td>
<td>221</td>
<td>V</td>
<td>vmsumuhm</td>
<td>Vector Multiply-Sum Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000027</td>
<td>V</td>
<td>222</td>
<td>V</td>
<td>vmsumuhs</td>
<td>Vector Multiply-Sum Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000028</td>
<td>V</td>
<td>220</td>
<td>V</td>
<td>vmsumshm</td>
<td>Vector Multiply-Sum Signed Halfword Modulo</td>
</tr>
<tr>
<td>VA 4</td>
<td>10000029</td>
<td>V</td>
<td>221</td>
<td>V</td>
<td>vmsumshs</td>
<td>Vector Multiply-Sum Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000040</td>
<td>V</td>
<td>203</td>
<td>V</td>
<td>vadduhm</td>
<td>Vector Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000042</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxuh</td>
<td>Vector Maximum Unsigned Halfword</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000044</td>
<td>V</td>
<td>240</td>
<td>V</td>
<td>vrlh</td>
<td>Vector Rotate Left Halfword</td>
</tr>
<tr>
<td>VC 4</td>
<td>10000046</td>
<td>V</td>
<td>233</td>
<td>V</td>
<td>vcmpequh</td>
<td>Vector Compare Equal To Unsigned Halfword</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000048</td>
<td>V</td>
<td>215</td>
<td>V</td>
<td>vmulouh</td>
<td>Vector Multiply Odd Unsigned Halfword</td>
</tr>
<tr>
<td>X 4</td>
<td>10000050</td>
<td>LMA</td>
<td>681</td>
<td>LMA</td>
<td>mulhwh</td>
<td>Multiply High Halfword to Word Signed</td>
</tr>
<tr>
<td>XO 4</td>
<td>10000058</td>
<td>LMA</td>
<td>677</td>
<td>LMA</td>
<td>machhw</td>
<td>Multiply Accumulate High Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>XO 4</td>
<td>10000058</td>
<td>LMA</td>
<td>677</td>
<td>LMA</td>
<td>machhwod</td>
<td>Multiply Accumulate High Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000080</td>
<td>V</td>
<td>204</td>
<td>V</td>
<td>vadduw</td>
<td>Vector Add Unsigned Word Modulo</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000082</td>
<td>V</td>
<td>229</td>
<td>V</td>
<td>vmaxxuw</td>
<td>Vector Maximum Unsigned Word</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000084</td>
<td>V</td>
<td>240</td>
<td>V</td>
<td>vrlw</td>
<td>Vector Rotate Left Word</td>
</tr>
<tr>
<td>VC 4</td>
<td>10000086</td>
<td>V</td>
<td>233</td>
<td>V</td>
<td>vcmpequw</td>
<td>Vector Compare Equal To Unsigned Word</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000088</td>
<td>V</td>
<td>216</td>
<td>V</td>
<td>vmulouw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000089</td>
<td>V</td>
<td>217</td>
<td>V</td>
<td>vmuluwm</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000098</td>
<td>LMA</td>
<td>678</td>
<td>LMA</td>
<td>machhwsu[]</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000098</td>
<td>LMA</td>
<td>678</td>
<td>LMA</td>
<td>machhwsuo[]</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000100</td>
<td>V</td>
<td>206</td>
<td>V</td>
<td>vadduqm</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000102</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxsb</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000104</td>
<td>V</td>
<td>241</td>
<td>V</td>
<td>vslb</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>1000110</td>
<td>LMA</td>
<td>680</td>
<td>LMA</td>
<td>mulchw[]</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000118</td>
<td>LMA</td>
<td>676</td>
<td>LMA</td>
<td>macchwu[]</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000118</td>
<td>LMA</td>
<td>676</td>
<td>LMA</td>
<td>macchwu[]</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000140</td>
<td>V</td>
<td>206</td>
<td>V</td>
<td>vaddcuq</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000142</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxsh</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000144</td>
<td>V</td>
<td>241</td>
<td>V</td>
<td>vslh</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000148</td>
<td>V</td>
<td>215</td>
<td>V</td>
<td>vmulosh</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>1000150</td>
<td>LMA</td>
<td>680</td>
<td>LMA</td>
<td>mulchw[]</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000158</td>
<td>LMA</td>
<td>675</td>
<td>LMA</td>
<td>macchwu[]</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000158</td>
<td>LMA</td>
<td>675</td>
<td>LMA</td>
<td>macchwo[]</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000180</td>
<td>V</td>
<td>202</td>
<td>V</td>
<td>vaddcuw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000182</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxsw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000184</td>
<td>V</td>
<td>241</td>
<td>V</td>
<td>vslw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000188</td>
<td>V</td>
<td>216</td>
<td>V</td>
<td>vmulosw</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000198</td>
<td>LMA</td>
<td>676</td>
<td>LMA</td>
<td>macchwsu[]</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>1000198</td>
<td>LMA</td>
<td>676</td>
<td>LMA</td>
<td>macchwsuo[]</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000200</td>
<td>SP</td>
<td>595</td>
<td>SP</td>
<td>evaddw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000200</td>
<td>V</td>
<td>205</td>
<td>V</td>
<td>vaddubs</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000202</td>
<td>SP</td>
<td>594</td>
<td>SP</td>
<td>evaddiw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000202</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vminub</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000204</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evsubfw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000204</td>
<td>V</td>
<td>242</td>
<td>V</td>
<td>vsrb</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000206</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evsubifw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000208</td>
<td>SP</td>
<td>594</td>
<td>SP</td>
<td>evabs</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000208</td>
<td>V</td>
<td>214</td>
<td>V</td>
<td>vmuleub</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000209</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evneg</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000211</td>
<td>SP</td>
<td>596</td>
<td>SP</td>
<td>evand</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000212</td>
<td>SP</td>
<td>596</td>
<td>SP</td>
<td>evandc</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000216</td>
<td>SP</td>
<td>639</td>
<td>SP</td>
<td>evxor</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000217</td>
<td>SP</td>
<td>632</td>
<td>SP</td>
<td>evor</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000218</td>
<td>SP</td>
<td>631</td>
<td>SP</td>
<td>evnor</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000219</td>
<td>SP</td>
<td>599</td>
<td>SP</td>
<td>eveqv</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000220</td>
<td>SP</td>
<td>635</td>
<td>SP</td>
<td>evsrwu</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000221</td>
<td>SP</td>
<td>635</td>
<td>SP</td>
<td>evsrws</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000222</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsrwu[]</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000223</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evsrw[]</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000224</td>
<td>SP</td>
<td>634</td>
<td>SP</td>
<td>evstw</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000226</td>
<td>634</td>
<td>SP</td>
<td>evslwi</td>
<td>Vector Shift Left Word Immediate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000228</td>
<td>632</td>
<td>SP</td>
<td>evrlw</td>
<td>Vector Rotate Left Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000229</td>
<td>634</td>
<td>SP</td>
<td>evsplati</td>
<td>Vector Splat Immediate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000230</td>
<td>597</td>
<td>SP</td>
<td>evcmpgtu</td>
<td>Vector Compare Greater Than Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000231</td>
<td>596</td>
<td>SP</td>
<td>evcmpgts</td>
<td>Vector Compare Greater Than Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000232</td>
<td>597</td>
<td>SP</td>
<td>evcmpltu</td>
<td>Vector Compare Less Than Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000233</td>
<td>597</td>
<td>SP</td>
<td>evcmplts</td>
<td>Vector Compare Less Than Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000234</td>
<td>596</td>
<td>SP</td>
<td>evcmpeq</td>
<td>Vector Compare Equal</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000240</td>
<td>205</td>
<td>V</td>
<td>vadduh</td>
<td>Vector Add Unsigned Halfword Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000242</td>
<td>230</td>
<td>V</td>
<td>vminuh</td>
<td>Vector Minimum Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000244</td>
<td>242</td>
<td>V</td>
<td>vsrh</td>
<td>Vector Shift Right Halfword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4 10000246</td>
<td>236</td>
<td>V</td>
<td>vcmpgtuh</td>
<td>Vector Compare Greater Than Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000248</td>
<td>215</td>
<td>V</td>
<td>vmuleuh</td>
<td>Vector Multiply Even Unsigned Halfword</td>
<td></td>
</tr>
<tr>
<td>EVS</td>
<td>4 10000278</td>
<td>633</td>
<td>SP</td>
<td>evsel</td>
<td>Vector Select</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000280</td>
<td>646</td>
<td>SP.FV</td>
<td>evfsadd</td>
<td>Vector Floating-Point Add</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000280</td>
<td>205</td>
<td>V</td>
<td>vadduw</td>
<td>Vector Add Unsigned Word Saturate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000281</td>
<td>646</td>
<td>SP.FV</td>
<td>evfssub</td>
<td>Vector Floating-Point Subtract</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000282</td>
<td>231</td>
<td>V</td>
<td>vminuw</td>
<td>Vector Minimum Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000284</td>
<td>242</td>
<td>V</td>
<td>vsrw</td>
<td>Vector Shift Right Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000285</td>
<td>645</td>
<td>SP.FV</td>
<td>evfsabs</td>
<td>Vector Floating-Point Absolute Value</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000286</td>
<td>645</td>
<td>SP.FV</td>
<td>evfsneg</td>
<td>Vector Floating-Point Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4 10000286</td>
<td>237</td>
<td>V</td>
<td>vcmpgtuw</td>
<td>Vector Compare Greater Than Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000288</td>
<td>646</td>
<td>SP.FV</td>
<td>evfsmul</td>
<td>Vector Floating-Point Multiply</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000288</td>
<td>216</td>
<td>V</td>
<td>vmuleuw</td>
<td>Vector Multiply Even Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000289</td>
<td>646</td>
<td>SP.FV</td>
<td>evfsdiv</td>
<td>Vector Floating-Point Divide</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000290</td>
<td>650</td>
<td>SP.FV</td>
<td>evfscfui</td>
<td>Vector Convert Floating-Point from Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000291</td>
<td>650</td>
<td>SP.FV</td>
<td>evfscfis</td>
<td>Vector Convert Floating-Point from Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000292</td>
<td>650</td>
<td>SP.FV</td>
<td>evfscfuf</td>
<td>Vector Convert Floating-Point from Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000293</td>
<td>650</td>
<td>SP.FV</td>
<td>evfscfsf</td>
<td>Vector Convert Floating-Point from Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000294</td>
<td>651</td>
<td>SP.FV</td>
<td>evfscui</td>
<td>Vector Convert Floating-Point to Unsigned Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000295</td>
<td>651</td>
<td>SP.FV</td>
<td>evfscuie</td>
<td>Vector Convert Floating-Point to Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000296</td>
<td>652</td>
<td>SP.FV</td>
<td>evfscuuf</td>
<td>Vector Convert Floating-Point to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000297</td>
<td>652</td>
<td>SP.FV</td>
<td>evfscstf</td>
<td>Vector Convert Floating-Point to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000298</td>
<td>651</td>
<td>SP.FV</td>
<td>evfscui</td>
<td>Vector Convert Floating-Point to Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000300</td>
<td>600</td>
<td>SP</td>
<td>evlidx</td>
<td>Vector Load Double Word into Double Word Indexed</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000300</td>
<td>202</td>
<td>V</td>
<td>vaddsb</td>
<td>Vector Add Signed Byte Saturate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000301</td>
<td>600</td>
<td>SP</td>
<td>evldd</td>
<td>Vector Load Double Word into Double Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000302</td>
<td>601</td>
<td>SP</td>
<td>evldwx</td>
<td>Vector Load Double into Two Words Indexed</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000302</td>
<td>230</td>
<td>V</td>
<td>vminsib</td>
<td>Vector Minimum Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000303</td>
<td>601</td>
<td>SP</td>
<td>evldw</td>
<td>Vector Load Double into Two Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000304</td>
<td>600</td>
<td>SP</td>
<td>evldhx</td>
<td>Vector Load Double into Four Half Words Indexed</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000304</td>
<td>243</td>
<td>V</td>
<td>vsrab</td>
<td>Vector Shift Right Algebraic Byte</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000305</td>
<td>SP</td>
<td></td>
<td>600</td>
<td>EVLDH</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000306</td>
<td>V</td>
<td></td>
<td>234</td>
<td>VCMPGTBSB</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000308</td>
<td>SP</td>
<td></td>
<td>601</td>
<td>EVLHESPLTX</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000308</td>
<td>V</td>
<td></td>
<td>214</td>
<td>VMULESB</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000309</td>
<td>SP</td>
<td></td>
<td>601</td>
<td>EVLHESPLAT</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000310</td>
<td>SP</td>
<td></td>
<td>603</td>
<td>EVLWHE</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000310</td>
<td>V</td>
<td></td>
<td>681</td>
<td>MULHWU</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000311</td>
<td>SP</td>
<td></td>
<td>603</td>
<td>EVLWHE</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000314</td>
<td>SP</td>
<td></td>
<td>604</td>
<td>EVLWHOUX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000315</td>
<td>SP</td>
<td></td>
<td>604</td>
<td>EVLWHOU</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000316</td>
<td>SP</td>
<td></td>
<td>603</td>
<td>EVLWHOSX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000317</td>
<td>SP</td>
<td></td>
<td>603</td>
<td>EVLWHOS</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000318</td>
<td>SP</td>
<td></td>
<td>605</td>
<td>EVLWWSPLAT</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000318</td>
<td>LMA</td>
<td></td>
<td>680</td>
<td>MACLHWU</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000318</td>
<td>LMA</td>
<td></td>
<td>680</td>
<td>MACLHWUO</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000319</td>
<td>SP</td>
<td></td>
<td>605</td>
<td>EVLWWSPLAT</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000320</td>
<td>SP</td>
<td></td>
<td>635</td>
<td>EVSTDX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000321</td>
<td>SP</td>
<td></td>
<td>635</td>
<td>EVSTTD</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000322</td>
<td>SP</td>
<td></td>
<td>636</td>
<td>EVSTDXW</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000323</td>
<td>SP</td>
<td></td>
<td>636</td>
<td>EVSTDW</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000324</td>
<td>SP</td>
<td></td>
<td>636</td>
<td>EVSTDHX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000325</td>
<td>SP</td>
<td></td>
<td>636</td>
<td>EVSTDH</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000330</td>
<td>SP</td>
<td></td>
<td>637</td>
<td>EVSTWHEX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000331</td>
<td>SP</td>
<td></td>
<td>637</td>
<td>EVSTWHE</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000334</td>
<td>SP</td>
<td></td>
<td>637</td>
<td>EVSTWHOX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000335</td>
<td>SP</td>
<td></td>
<td>637</td>
<td>EVSTWHO</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000338</td>
<td>SP</td>
<td></td>
<td>637</td>
<td>EVSTWWEX</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000339</td>
<td>SP</td>
<td></td>
<td>637</td>
<td>EVSTWWE</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000340</td>
<td>V</td>
<td></td>
<td>202</td>
<td>VMADDH</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000342</td>
<td>V</td>
<td></td>
<td>230</td>
<td>VMINSH</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000344</td>
<td>V</td>
<td></td>
<td>243</td>
<td>VSRAH</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000346</td>
<td>V</td>
<td></td>
<td>234</td>
<td>VCMPGTSH</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000348</td>
<td>V</td>
<td></td>
<td>215</td>
<td>VMULESH</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>10000350</td>
<td>LMA</td>
<td></td>
<td>681</td>
<td>MULHWU</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000358</td>
<td>LMA</td>
<td></td>
<td>679</td>
<td>MACLHWU</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>10000358</td>
<td>LMA</td>
<td></td>
<td>679</td>
<td>MACLHWUO</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000380</td>
<td>V</td>
<td></td>
<td>203</td>
<td>VADDSSWS</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000382</td>
<td>V</td>
<td></td>
<td>231</td>
<td>VMINSW</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000384</td>
<td>V</td>
<td></td>
<td>243</td>
<td>VSRAW</td>
</tr>
<tr>
<td>Format</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>VC</td>
<td>10000386</td>
<td>V</td>
<td></td>
<td>235</td>
<td>V</td>
<td>vcmptsw</td>
</tr>
<tr>
<td>VX</td>
<td>10000388</td>
<td>V</td>
<td></td>
<td>216</td>
<td>V</td>
<td>vmuelsw</td>
</tr>
<tr>
<td>XO</td>
<td>10000398</td>
<td>LMA</td>
<td></td>
<td>680</td>
<td>LMA</td>
<td>mclhwus</td>
</tr>
<tr>
<td>XO</td>
<td>10000398</td>
<td>LMA</td>
<td></td>
<td>680</td>
<td>LMA</td>
<td>mclhwusuo</td>
</tr>
<tr>
<td>VX</td>
<td>10000400</td>
<td>V</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsuubm</td>
</tr>
<tr>
<td>VX</td>
<td>10000402</td>
<td>V</td>
<td></td>
<td>227</td>
<td>V</td>
<td>vavgub</td>
</tr>
<tr>
<td>EVX</td>
<td>10000403</td>
<td>SP</td>
<td></td>
<td>610</td>
<td>SP</td>
<td>evmhessf</td>
</tr>
<tr>
<td>VX</td>
<td>10000404</td>
<td>V</td>
<td></td>
<td>238</td>
<td>V</td>
<td>vand</td>
</tr>
<tr>
<td>EVX</td>
<td>10000407</td>
<td>SP</td>
<td></td>
<td>619</td>
<td>SP</td>
<td>evmhossf</td>
</tr>
<tr>
<td>EVX</td>
<td>10000408</td>
<td>SP</td>
<td></td>
<td>613</td>
<td>SP</td>
<td>evmheumia</td>
</tr>
<tr>
<td>VX</td>
<td>10000408</td>
<td>V</td>
<td></td>
<td>259</td>
<td>V</td>
<td>vpmsumb</td>
</tr>
<tr>
<td>EVX</td>
<td>10000409</td>
<td>SP</td>
<td></td>
<td>609</td>
<td>SP</td>
<td>evmhesm</td>
</tr>
<tr>
<td>EVX</td>
<td>10000423</td>
<td>SP</td>
<td></td>
<td>610</td>
<td>SP</td>
<td>evmhessfa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000427</td>
<td>SP</td>
<td></td>
<td>619</td>
<td>SP</td>
<td>evmhosfa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000428</td>
<td>SP</td>
<td></td>
<td>613</td>
<td>SP</td>
<td>evmheumia</td>
</tr>
<tr>
<td>EVX</td>
<td>10000429</td>
<td>SP</td>
<td></td>
<td>609</td>
<td>SP</td>
<td>evmhesm</td>
</tr>
<tr>
<td>VX</td>
<td>10000440</td>
<td>V</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsuubuhm</td>
</tr>
<tr>
<td>VX</td>
<td>10000442</td>
<td>V</td>
<td></td>
<td>227</td>
<td>V</td>
<td>vavguh</td>
</tr>
<tr>
<td>VX</td>
<td>10000444</td>
<td>V</td>
<td></td>
<td>238</td>
<td>V</td>
<td>vandc</td>
</tr>
<tr>
<td>EVX</td>
<td>10000447</td>
<td>SP</td>
<td></td>
<td>624</td>
<td>SP</td>
<td>evmwhssf</td>
</tr>
<tr>
<td>VX</td>
<td>10000448</td>
<td>V</td>
<td></td>
<td>260</td>
<td>V</td>
<td>vpmsumh</td>
</tr>
<tr>
<td>EVX</td>
<td>10000453</td>
<td>SP</td>
<td></td>
<td>629</td>
<td>SP</td>
<td>evmwssf</td>
</tr>
<tr>
<td>EVX</td>
<td>10000458</td>
<td>SP</td>
<td></td>
<td>630</td>
<td>SP</td>
<td>evmwumi</td>
</tr>
<tr>
<td>EVX</td>
<td>10000459</td>
<td>SP</td>
<td></td>
<td>628</td>
<td>SP</td>
<td>evmwsmi</td>
</tr>
<tr>
<td>EVX</td>
<td>10000467</td>
<td>SP</td>
<td></td>
<td>624</td>
<td>SP</td>
<td>evmwhssfa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000468</td>
<td>SP</td>
<td></td>
<td>626</td>
<td>SP</td>
<td>evmwulumia</td>
</tr>
<tr>
<td>EVX</td>
<td>10000473</td>
<td>SP</td>
<td></td>
<td>629</td>
<td>SP</td>
<td>evmwssfa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000478</td>
<td>SP</td>
<td></td>
<td>630</td>
<td>SP</td>
<td>evmwumia</td>
</tr>
<tr>
<td>EVX</td>
<td>10000479</td>
<td>SP</td>
<td></td>
<td>628</td>
<td>SP</td>
<td>evmwsmia</td>
</tr>
<tr>
<td>VX</td>
<td>10000480</td>
<td>V</td>
<td></td>
<td>210</td>
<td>V</td>
<td>vsuubuwm</td>
</tr>
<tr>
<td>VX</td>
<td>10000482</td>
<td>V</td>
<td></td>
<td>227</td>
<td>V</td>
<td>vavgw</td>
</tr>
<tr>
<td>VX</td>
<td>10000484</td>
<td>V</td>
<td></td>
<td>239</td>
<td>V</td>
<td>vor</td>
</tr>
<tr>
<td>VX</td>
<td>10000488</td>
<td>V</td>
<td></td>
<td>260</td>
<td>V</td>
<td>vpmsumw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000500</td>
<td>SP</td>
<td></td>
<td>614</td>
<td>SP</td>
<td>evmheusiaaw</td>
</tr>
<tr>
<td>VX</td>
<td>10000500</td>
<td>V</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsuubuqm</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000501</td>
<td>SP</td>
<td>evmhessiaaw</td>
<td>612</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000502</td>
<td>V</td>
<td>vavgsb</td>
<td>226</td>
<td>Vector Average Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000503</td>
<td>SP</td>
<td>evmhessfaaw</td>
<td>611</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000504</td>
<td>SP</td>
<td>evmhousiaaw</td>
<td>622</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000504</td>
<td>V</td>
<td>vnor</td>
<td>239</td>
<td>Vector Logical NOR</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000505</td>
<td>SP</td>
<td>evmhossiaaw</td>
<td>621</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000507</td>
<td>SP</td>
<td>evmhossfaaw</td>
<td>620</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000508</td>
<td>SP</td>
<td>evmheumiaaw</td>
<td>613</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000508</td>
<td>256</td>
<td>V.Crypto vcipher</td>
<td>609</td>
<td>Vector AES Cipher</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000509</td>
<td>256</td>
<td>V.Crypto vcipheriaaw</td>
<td>609</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000509</td>
<td>SP</td>
<td>evmhesmiaaw</td>
<td>626</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000509</td>
<td>212</td>
<td>V</td>
<td>vsbcuq</td>
<td>Vector Subtract &amp; write Carry Unsigned Quadword</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000510</td>
<td>SP</td>
<td>evmwssiaaw</td>
<td>625</td>
<td>Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000542</td>
<td>226</td>
<td>V</td>
<td>vavgsh</td>
<td>Vector Average Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000544</td>
<td>239</td>
<td>V</td>
<td>vorc</td>
<td>Vector OR with Complement</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000548</td>
<td>SP</td>
<td>evmwlusiaaw</td>
<td>626</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000548</td>
<td>257</td>
<td>V.Crypto</td>
<td>257</td>
<td>Vector AES Inverse Cipher</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000549</td>
<td>SP</td>
<td>evmwcsiaw</td>
<td>625</td>
<td>Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000549</td>
<td>257</td>
<td>V.Crypto</td>
<td>257</td>
<td>Vector AES Inverse Cipher Last</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000553</td>
<td>SP</td>
<td>evmwssfaa</td>
<td>629</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000559</td>
<td>SP</td>
<td>evmwsmiaaw</td>
<td>628</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000560</td>
<td>SP</td>
<td>evmheusianw</td>
<td>614</td>
<td>Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000560</td>
<td>208</td>
<td>V</td>
<td>vsbcuw</td>
<td>Vector Subtract and Write Carry-Out Unsigned Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000581</td>
<td>SP</td>
<td>evmhessianw</td>
<td>612</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 10000582</td>
<td>226</td>
<td>V</td>
<td>vavgsb</td>
<td>Vector Average Signed Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000583</td>
<td>SP</td>
<td>evmhessfanaaw</td>
<td>611</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>EVX</td>
<td>4</td>
<td>10000584</td>
<td>622</td>
<td>SP</td>
<td>evmhousianw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000584</td>
<td>V</td>
<td>238</td>
<td>Vnand</td>
<td>vnand</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000585</td>
<td>SP</td>
<td>621</td>
<td>evmhossianw</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000587</td>
<td>SP</td>
<td>620</td>
<td>evmhosssfanw</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000588</td>
<td>SP</td>
<td>613</td>
<td>evmheumianw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000589</td>
<td>SP</td>
<td>609</td>
<td>evmhesianw</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000600</td>
<td>V</td>
<td>211</td>
<td>vsububs</td>
<td>vsububs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000604</td>
<td>Mvscr</td>
<td>268</td>
<td>Mvscr</td>
<td>Mvscr</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000608</td>
<td>V</td>
<td>225</td>
<td>vsum4ubs</td>
<td>vsum4ubs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000640</td>
<td>V</td>
<td>210</td>
<td>vsubuhhs</td>
<td>vsubuhhs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000644</td>
<td>Mtvsr</td>
<td>268</td>
<td>Mtvsr</td>
<td>Mtvsr</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000648</td>
<td>V</td>
<td>224</td>
<td>vsum4shs</td>
<td>vsum4shs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000680</td>
<td>V</td>
<td>211</td>
<td>vsubuws</td>
<td>vsubuws</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000682</td>
<td>Vcrypt</td>
<td>258</td>
<td>Yshasigmaw</td>
<td>Yshasigmaw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000684</td>
<td>V</td>
<td>238</td>
<td>Veqy</td>
<td>Veqy</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000688</td>
<td>V</td>
<td>223</td>
<td>Vsum2sws</td>
<td>Vsum2sws</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000700</td>
<td>V</td>
<td>208</td>
<td>Vsubbsbs</td>
<td>Vsubbsbs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000702</td>
<td>Vc1zb</td>
<td>263</td>
<td>Vc1zb</td>
<td>Vc1zb</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000703</td>
<td>Vpopcntb</td>
<td>264</td>
<td>Vpopcntb</td>
<td>Vpopcntb</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000708</td>
<td>V</td>
<td>224</td>
<td>Vsum4sbs</td>
<td>Vsum4sbs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000740</td>
<td>V</td>
<td>208</td>
<td>Vsubshs</td>
<td>Vsubshs</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000742</td>
<td>Vc1zh</td>
<td>263</td>
<td>Vc1zh</td>
<td>Vc1zh</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000743</td>
<td>Vpopcnth</td>
<td>264</td>
<td>Vpopcnth</td>
<td>Vpopcnth</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000780</td>
<td>V</td>
<td>209</td>
<td>Vsubsws</td>
<td>Vsubsws</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000782</td>
<td>Vc1zw</td>
<td>263</td>
<td>Vc1zw</td>
<td>Vc1zw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000783</td>
<td>Vpopcntw</td>
<td>264</td>
<td>Vpopcntw</td>
<td>Vpopcntw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000788</td>
<td>V</td>
<td>223</td>
<td>Vsumsws</td>
<td>Vsumsws</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000000A</td>
<td>Vvaddfp</td>
<td>244</td>
<td>Vvaddfp</td>
<td>Vvaddfp</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000000C</td>
<td>Vmrgb</td>
<td>194</td>
<td>Vmrgb</td>
<td>Vmrgb</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000000E</td>
<td>Vpkhuw</td>
<td>190</td>
<td>Vpkhuw</td>
<td>Vpkhuw</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002A</td>
<td>Vsel</td>
<td>199</td>
<td>Vsel</td>
<td>Vsel</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002B</td>
<td>Vperm</td>
<td>198</td>
<td>Vperm</td>
<td>Vperm</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002C</td>
<td>Vsldoi</td>
<td>200</td>
<td>Vsldoi</td>
<td>Vsldoi</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002D</td>
<td>Vraidd</td>
<td>261</td>
<td>Vraidd</td>
<td>Vraidd</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002E</td>
<td>Vmnaddfp</td>
<td>245</td>
<td>Vmnaddfp</td>
<td>Vmnaddfp</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000002F</td>
<td>Vmnsubfp</td>
<td>245</td>
<td>Vmnsubfp</td>
<td>Vmnsubfp</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000003C</td>
<td>Vaddeuqm</td>
<td>206</td>
<td>Vaddeuqm</td>
<td>Vaddeuqm</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000003D</td>
<td>Vaddecuq</td>
<td>206</td>
<td>Vaddecuq</td>
<td>Vaddecuq</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>VA 4</td>
<td>100003E</td>
<td>V</td>
<td></td>
<td>212</td>
<td>V</td>
<td>Vector Subtract Extended Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>VA 4</td>
<td>100003F</td>
<td>V</td>
<td></td>
<td>212</td>
<td>V</td>
<td>Vector Subtract Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>VX 4</td>
<td>100004A</td>
<td>V</td>
<td></td>
<td>244</td>
<td>V</td>
<td>Vector Subtract Single-Precision</td>
</tr>
<tr>
<td>VX 4</td>
<td>100004C</td>
<td>V</td>
<td></td>
<td>194</td>
<td>V</td>
<td>Vector Merge High Halfword</td>
</tr>
<tr>
<td>VX 4</td>
<td>100004E</td>
<td>V</td>
<td></td>
<td>191</td>
<td>V</td>
<td>Vector Pack Signed Word Signed Modulo</td>
</tr>
<tr>
<td>VO 4</td>
<td>100005C</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td>LMA</td>
<td>Negative Multiply Accumulate High Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>VO 4</td>
<td>100005C</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td>LMA</td>
<td>Negative Multiply Accumulate High Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>VX 4</td>
<td>100008C</td>
<td>V</td>
<td></td>
<td>195</td>
<td>V</td>
<td>Vector Merge High Word</td>
</tr>
<tr>
<td>VX 4</td>
<td>100008E</td>
<td>V</td>
<td></td>
<td>191</td>
<td>V</td>
<td>Vector Add Signed Halfword Signed Saturate</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000C0</td>
<td>V</td>
<td></td>
<td>203</td>
<td>V</td>
<td>Vector Add Unsinged Doubleword Modulo</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000C2</td>
<td>V</td>
<td></td>
<td>228</td>
<td>V</td>
<td>Vector Maximum Unsinged Doubleword</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000C4</td>
<td>V</td>
<td></td>
<td>240</td>
<td>V</td>
<td>Vector Rotate Left Doubleword</td>
</tr>
<tr>
<td>VC 4</td>
<td>10000C6</td>
<td>V</td>
<td></td>
<td>252</td>
<td>V</td>
<td>Vector Compare Equal To Single-Precision</td>
</tr>
<tr>
<td>VC 4</td>
<td>10000C7</td>
<td>V</td>
<td></td>
<td>233</td>
<td>V</td>
<td>Vector Compare Equal To Signed Doubleword</td>
</tr>
<tr>
<td>VX 4</td>
<td>10000CE</td>
<td>V</td>
<td></td>
<td>191</td>
<td>V</td>
<td>Vector Pack Unsinged Word Unsinged Saturate</td>
</tr>
<tr>
<td>VO 4</td>
<td>10000D8</td>
<td>LMA</td>
<td></td>
<td>677</td>
<td>LMA</td>
<td>Multiply Accumulate High Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>VO 4</td>
<td>10000D8</td>
<td>LMA</td>
<td></td>
<td>677</td>
<td>LMA</td>
<td>Multiply Accumulate High Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>VO 4</td>
<td>10000DC</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td>LMA</td>
<td>Negative Multiply Accumulate High Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>VO 4</td>
<td>10000DC</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td>LMA</td>
<td>Negative Multiply Accumulate High Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>VX 4</td>
<td>100010A</td>
<td>V</td>
<td></td>
<td>255</td>
<td>V</td>
<td>Vector Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>VX 4</td>
<td>100010C</td>
<td>V</td>
<td></td>
<td>194</td>
<td>V</td>
<td>Vector Merge Low Byte</td>
</tr>
<tr>
<td>VX 4</td>
<td>100010E</td>
<td>V</td>
<td></td>
<td>189</td>
<td>V</td>
<td>Vector Pack Signed Halfword Unsinged Saturate</td>
</tr>
<tr>
<td>VX 4</td>
<td>100014A</td>
<td>V</td>
<td></td>
<td>255</td>
<td>V</td>
<td>Vector Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>VX 4</td>
<td>100014C</td>
<td>V</td>
<td></td>
<td>194</td>
<td>V</td>
<td>Vector Merge Low Halfword</td>
</tr>
<tr>
<td>VX 4</td>
<td>100014E</td>
<td>V</td>
<td></td>
<td>190</td>
<td>V</td>
<td>Vector Pack Signed Word Unsinged Saturate</td>
</tr>
<tr>
<td>VO 4</td>
<td>100015C</td>
<td>LMA</td>
<td></td>
<td>682</td>
<td>LMA</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Modulo Signed</td>
</tr>
<tr>
<td>VO 4</td>
<td>100015C</td>
<td>LMA</td>
<td></td>
<td>682</td>
<td>LMA</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Modulo Signed &amp; record OV</td>
</tr>
<tr>
<td>VX 4</td>
<td>100018A</td>
<td>V</td>
<td></td>
<td>254</td>
<td>V</td>
<td>Vector 2 Raised to the Exponent Estimate Single-Precision</td>
</tr>
<tr>
<td>VX 4</td>
<td>100018C</td>
<td>V</td>
<td></td>
<td>195</td>
<td>V</td>
<td>Vector Merge Low Word</td>
</tr>
<tr>
<td>VX 4</td>
<td>100018E</td>
<td>V</td>
<td></td>
<td>188</td>
<td>V</td>
<td>Vector Pack Signed Halfword Signed Saturate</td>
</tr>
<tr>
<td>VX 4</td>
<td>10001C2</td>
<td>V</td>
<td></td>
<td>228</td>
<td>V</td>
<td>Vector Maximum Signed Doubleword</td>
</tr>
<tr>
<td>VX 4</td>
<td>10001C4</td>
<td>V</td>
<td></td>
<td>200</td>
<td>V</td>
<td>Vector Shift Left</td>
</tr>
<tr>
<td>VC 4</td>
<td>10001C6</td>
<td>V</td>
<td></td>
<td>252</td>
<td>V</td>
<td>Vector Compare Greater Than or Equal To Single-Precision</td>
</tr>
<tr>
<td>VX 4</td>
<td>10001CA</td>
<td>V</td>
<td></td>
<td>254</td>
<td>V</td>
<td>Vector Log Base 2 Estimate Single-Precision</td>
</tr>
<tr>
<td>VX 4</td>
<td>10001CE</td>
<td>V</td>
<td></td>
<td>189</td>
<td>V</td>
<td>Vector Pack Signed Word Signed Saturate</td>
</tr>
<tr>
<td>VO 4</td>
<td>10001D8</td>
<td>LMA</td>
<td></td>
<td>675</td>
<td>LMA</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>VO 4</td>
<td>10001D8</td>
<td>LMA</td>
<td></td>
<td>675</td>
<td>LMA</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>Format</td>
<td>Primary</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100001DC</td>
<td>682</td>
<td>LMA</td>
<td>nmacchws[,]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
</tr>
<tr>
<td>XO</td>
<td>4</td>
<td>100001DC</td>
<td>682</td>
<td>LMA</td>
<td>nmacchwso[,]</td>
<td>Negative Multiply Accumulate Cross Halfword to Word Saturate Signed &amp; record OV</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020A</td>
<td>599</td>
<td>SP</td>
<td>evextsb</td>
<td>Vector Extend Sign Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000020A</td>
<td>249</td>
<td>V</td>
<td>vrfi</td>
<td>Vector Round to Single-Precision Integer Nearest</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020B</td>
<td>599</td>
<td>SP</td>
<td>evextsh</td>
<td>Vector Extend Sign Half Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020C</td>
<td>633</td>
<td>SP</td>
<td>evrndw</td>
<td>Vector Round Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000020C</td>
<td>197</td>
<td>V</td>
<td>vspltb</td>
<td>Vector Splat Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020D</td>
<td>598</td>
<td>SP</td>
<td>evcntlzww</td>
<td>Vector Count Leading Zeros Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020E</td>
<td>598</td>
<td>SP</td>
<td>evcntisw</td>
<td>Vector Count Leading Signed Bits Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000020E</td>
<td>193</td>
<td>V</td>
<td>vupkhbsb</td>
<td>Vector Unpack High Signed Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000020F</td>
<td>594</td>
<td>SP</td>
<td>brinc</td>
<td>Bit Reversed Increment</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000021B</td>
<td>632</td>
<td>SP</td>
<td>evorc</td>
<td>Vector OR with Complement</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000021B</td>
<td>631</td>
<td>SP</td>
<td>evnand</td>
<td>Vector NAND</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022A</td>
<td>633</td>
<td>SP</td>
<td>evrlni</td>
<td>Vector Rotate Left Word Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022B</td>
<td>634</td>
<td>SP</td>
<td>evsplatfi</td>
<td>Vector Splat Fractional Immediate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022C</td>
<td>605</td>
<td>SP</td>
<td>evmergehi</td>
<td>Vector Merge High</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022D</td>
<td>605</td>
<td>SP</td>
<td>evmergeho</td>
<td>Vector Merge Low</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000022E</td>
<td>606</td>
<td>SP</td>
<td>evmergehlo</td>
<td>Vector Merge Low/High</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000024A</td>
<td>249</td>
<td>V</td>
<td>vrfiz</td>
<td>Vector Round to Single-Precision Integer toward Zero</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000024C</td>
<td>197</td>
<td>V</td>
<td>vsplth</td>
<td>Vector Splat Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000024E</td>
<td>193</td>
<td>V</td>
<td>vupkhsh</td>
<td>Vector Unpack High Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000028A</td>
<td>249</td>
<td>V</td>
<td>vrfip</td>
<td>Vector Round to Single-Precision Integer toward +Infinity</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000028C</td>
<td>647</td>
<td>SP.FV</td>
<td>evfscmpgt</td>
<td>Vector Floating-Point Compare Greater Than</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000028C</td>
<td>197</td>
<td>V</td>
<td>vspltw</td>
<td>Vector Splat Word</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000028D</td>
<td>647</td>
<td>SP.FV</td>
<td>evfscmplt</td>
<td>Vector Floating-Point Compare Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000028E</td>
<td>648</td>
<td>SP.FV</td>
<td>evfscmpeq</td>
<td>Vector Floating-Point Compare Equal</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000028E</td>
<td>193</td>
<td>V</td>
<td>vupkltsb</td>
<td>Vector Unpack Low Signed Byte</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029A</td>
<td>651</td>
<td>SP.FV</td>
<td>evfsctsiz</td>
<td>Vector Convert Floating-Point to Signed Integer with Round toward Zero</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029C</td>
<td>648</td>
<td>SP.FV</td>
<td>evfstdssgt</td>
<td>Vector Floating-Point Test Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029D</td>
<td>649</td>
<td>SP.FV</td>
<td>evfstdsllt</td>
<td>Vector Floating-Point Test Less Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029E</td>
<td>649</td>
<td>SP.FV</td>
<td>evfstdsteq</td>
<td>Vector Floating-Point Test Equal</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td>654</td>
<td>SP.FS</td>
<td>efsadd</td>
<td>Floating-Point Add</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C1</td>
<td>654</td>
<td>SP.FS</td>
<td>efssub</td>
<td>Floating-Point Subtract</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100002C2</td>
<td>230</td>
<td>V</td>
<td>vminud</td>
<td>Vector Minimum Unsigned Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C4</td>
<td>653</td>
<td>SP.FS</td>
<td>efsabs</td>
<td>Floating-Point Absolute Value</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100002C4</td>
<td>201</td>
<td>V</td>
<td>vshr</td>
<td>Vector Shift Right</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C5</td>
<td>653</td>
<td>SP.FS</td>
<td>efsnabs</td>
<td>Floating-Point Negative Absolute Value</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C6</td>
<td>653</td>
<td>SP.FS</td>
<td>efsneg</td>
<td>Floating-Point Negate</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100002C6</td>
<td>253</td>
<td>V</td>
<td>vcmptgtf[.]</td>
<td>Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100002C7</td>
<td>236</td>
<td>V</td>
<td>vcmptgud[.]</td>
<td>Vector Compare Greater Than Unsigned Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C8</td>
<td>654</td>
<td>SP.FS</td>
<td>efsmul</td>
<td>Floating-Point Multiply</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002C9</td>
<td>654</td>
<td>SP.FS</td>
<td>efsdiv</td>
<td>Floating-Point Divide</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100002CA</td>
<td>250</td>
<td>V</td>
<td>vrfim</td>
<td>Vector Round to Single-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CC</td>
<td>655</td>
<td>SP.FS</td>
<td>efscmpeq</td>
<td>Floating-Point Compare Greater Than</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100002CD</td>
<td>655</td>
<td>SP.FS</td>
<td>efscmpeql</td>
<td>Floating-Point Compare Less Than</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002CE</td>
<td></td>
<td></td>
<td>656</td>
<td>SP,FS</td>
<td>efscmpeq Floating-Point Compare Equal</td>
</tr>
<tr>
<td>VX 4</td>
<td>100002CE</td>
<td>V</td>
<td></td>
<td>193</td>
<td>V</td>
<td>yvplsh Vector Unpack Low Signed Halfword</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002CF</td>
<td>SP,FD</td>
<td></td>
<td>667</td>
<td></td>
<td>efscfd Floating-Point Single-Precision Convert from Double-Precision</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D0</td>
<td>SP,FS</td>
<td></td>
<td>658</td>
<td></td>
<td>efscfui Convert Floating-Point from Unsigned Integer</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D1</td>
<td>SP,FS</td>
<td></td>
<td>658</td>
<td></td>
<td>efscfsi Convert Floating-Point from Signed Integer</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D2</td>
<td>SP,FS</td>
<td></td>
<td>658</td>
<td></td>
<td>efscfuf Convert Floating-Point from Unsigned Fraction</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D3</td>
<td>SP,FS</td>
<td></td>
<td>658</td>
<td></td>
<td>efscfsf Convert Floating-Point from Signed Fraction</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D4</td>
<td>SP,FS</td>
<td></td>
<td>658</td>
<td></td>
<td>efscstui Convert Floating-Point to Signed Integer</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D5</td>
<td>SP,FS</td>
<td></td>
<td>658</td>
<td></td>
<td>efscstsi Convert Floating-Point to Signed Integer</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D6</td>
<td>SP,FS</td>
<td></td>
<td>659</td>
<td></td>
<td>efscstuf Convert Floating-Point to Unsigned Fraction</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D7</td>
<td>SP,FS</td>
<td></td>
<td>659</td>
<td></td>
<td>efscstsf Convert Floating-Point to Signed Fraction</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002D8</td>
<td>SP,FS</td>
<td></td>
<td>659</td>
<td></td>
<td>efscstuiz Convert Floating-Point to Signed Integer with Round toward Zero</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002DA</td>
<td>SP,FS</td>
<td></td>
<td>659</td>
<td></td>
<td>efscstsiz Convert Floating-Point to Signed Integer with Round toward Zero</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002DC</td>
<td>SP,FS</td>
<td></td>
<td>656</td>
<td></td>
<td>efststgt Floating-Point Test Greater Than</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002DD</td>
<td>SP,FS</td>
<td></td>
<td>657</td>
<td></td>
<td>efststlt Floating-Point Test Less Than</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002DE</td>
<td>SP,FS</td>
<td></td>
<td>657</td>
<td></td>
<td>efststeq Floating-Point Test Equal</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E0</td>
<td>SP,FD</td>
<td></td>
<td>661</td>
<td></td>
<td>efddadd Floating-Point Double-Precision Add</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E1</td>
<td>SP,FD</td>
<td></td>
<td>661</td>
<td></td>
<td>efdsdub Floating-Point Double-Precision Subtract</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E2</td>
<td>SP,FD</td>
<td></td>
<td>664</td>
<td></td>
<td>efdfu Convert Floating-Point Double-Precision from Integer Doubleword</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E3</td>
<td>SP,FD</td>
<td></td>
<td>664</td>
<td></td>
<td>efdsf Convert Floating-Point Double-Precision to Integer Doubleword</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E4</td>
<td>SP,FD</td>
<td></td>
<td>660</td>
<td></td>
<td>efdb Convert Floating-Point Double-Precision Absolute Value</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E5</td>
<td>SP,FD</td>
<td></td>
<td>660</td>
<td></td>
<td>efdn Convert Floating-Point Double-Precision Negative Absolute Value</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E6</td>
<td>SP,FD</td>
<td></td>
<td>660</td>
<td></td>
<td>efd Convert Floating-Point Double-Precision Negate</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E8</td>
<td>SP,FD</td>
<td></td>
<td>661</td>
<td></td>
<td>efdmul Convert Floating-Point Double-Precision Multiply</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002E9</td>
<td>SP,FD</td>
<td></td>
<td>661</td>
<td></td>
<td>efddiv Convert Floating-Point Double-Precision Divide</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002EA</td>
<td>SP,FD</td>
<td></td>
<td>665</td>
<td></td>
<td>efdt Convert Double-Precision to Integer Doubleword with Round toward Zero</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002EB</td>
<td>SP,FD</td>
<td></td>
<td>665</td>
<td></td>
<td>efdts Convert Double-Precision to Signed Integer Doubleword with Round toward Zero</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002EC</td>
<td>SP,FD</td>
<td></td>
<td>662</td>
<td></td>
<td>efdcmp Convert Double-Precision Compare Greater Than</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002ED</td>
<td>SP,FD</td>
<td></td>
<td>662</td>
<td></td>
<td>efdcmp Convert Double-Precision Compare Less Than</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002EE</td>
<td>SP,FD</td>
<td></td>
<td>662</td>
<td></td>
<td>efdcm Convert Double-Precision Compare Equal</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002EF</td>
<td>SP,FD</td>
<td></td>
<td>666</td>
<td></td>
<td>efdc Convert Double-Precision Convert from Single-Precision</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002F0</td>
<td>SP,FD</td>
<td></td>
<td>663</td>
<td></td>
<td>efdc Convert Floating-Point Double-Precision from Integer</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002F1</td>
<td>SP,FD</td>
<td></td>
<td>663</td>
<td></td>
<td>efdc Convert Floating-Point Double-Precision from Signed Integer</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002F2</td>
<td>SP,FD</td>
<td></td>
<td>664</td>
<td></td>
<td>efdc Convert Floating-Point Double-Precision from Unsigned Fraction</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002F3</td>
<td>SP,FD</td>
<td></td>
<td>664</td>
<td></td>
<td>efdc Convert Floating-Point Double-Precision from Signed Fraction</td>
</tr>
<tr>
<td>EVX 4</td>
<td>100002F4</td>
<td>SP,FD</td>
<td></td>
<td>664</td>
<td></td>
<td>efdc Convert Floating-Point Double-Precision to Unsigned Integer</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002F5</td>
<td>664</td>
<td>SP.FD</td>
<td>ef dctsi</td>
<td>Convert Floating-Point Double-Precision to Signed Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002F6</td>
<td>666</td>
<td>SP.FD</td>
<td>ef dctuf</td>
<td>Convert Floating-Point Double-Precision to Unsigned Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002F7</td>
<td>666</td>
<td>SP.FD</td>
<td>ef dctsf</td>
<td>Convert Floating-Point Double-Precision to Signed Fraction</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002F8</td>
<td>666</td>
<td>SP.FD</td>
<td>ef dctuiz</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002FA</td>
<td>666</td>
<td>SP.FD</td>
<td>ef dctsz</td>
<td>Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002FC</td>
<td>662</td>
<td>SP.FD</td>
<td>ef dstgt</td>
<td>Floating-Point Double-Precision Test Greater Than</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002FE</td>
<td>663</td>
<td>SP.FD</td>
<td>ef dstteq</td>
<td>Floating-Point Double-Precision Test Equal</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000030A</td>
<td>248</td>
<td>V</td>
<td>vcfux</td>
<td>Vector Convert From Unsigned Fixed-Point Word To Single-Precision</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000030C</td>
<td>602</td>
<td>SP</td>
<td>evlh housplatx</td>
<td>Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000030C</td>
<td>198</td>
<td>V</td>
<td>vspltisb</td>
<td>Vector Splat Immediate Signed Byte</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000030D</td>
<td>602</td>
<td>SP</td>
<td>evlh housplat</td>
<td>Vector Load Half Word into Half Word Odd Unsigned and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000030E</td>
<td>602</td>
<td>SP</td>
<td>evlh housplax</td>
<td>Vector Load Half Word into Half Word Odd Signed and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000030E</td>
<td>187</td>
<td>V</td>
<td>vpkpx</td>
<td>Vector Pack Pixel</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000030F</td>
<td>602</td>
<td>SP</td>
<td>evlh ossplat</td>
<td>Vector Load Half Word into Half Word Odd Signed and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000031C</td>
<td>604</td>
<td>SP</td>
<td>evlw hsplax</td>
<td>Vector Load Word into Two Half Words and Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000031D</td>
<td>604</td>
<td>SP</td>
<td>evlw splat</td>
<td>Vector Load Word into Two Half Words and Splat</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000033C</td>
<td>638</td>
<td>SP</td>
<td>evstwwox</td>
<td>Vector Store Word of Word from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000033D</td>
<td>638</td>
<td>SP</td>
<td>evst wwo</td>
<td>Vector Store Word of Word from Odd</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000034A</td>
<td>248</td>
<td>V</td>
<td>vcf sx</td>
<td>Vector Convert From Signed Fixed-Point Word To Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000034C</td>
<td>198</td>
<td>V</td>
<td>vspltish</td>
<td>Vector Splat Immediate Signed Halfword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000034E</td>
<td>190</td>
<td>V</td>
<td>vupkhpx</td>
<td>Vector Unpack High Pixel</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 1000035C</td>
<td>684</td>
<td>LMA</td>
<td>nmach lw[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Modulo Signed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 1000035C</td>
<td>684</td>
<td>LMA</td>
<td>nmach lwo[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Modulo Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000038A</td>
<td>247</td>
<td>V</td>
<td>vctuxs</td>
<td>Vector Convert From Single-Precision To Unsigned Fixed-Point Word Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000038C</td>
<td>198</td>
<td>V</td>
<td>vspltisw</td>
<td>Vector Splat Immediate Signed Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>4 100003C2</td>
<td>230</td>
<td>V</td>
<td>vmins d</td>
<td>Vector Minimum Signed Doubleword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100003C4</td>
<td>243</td>
<td>V</td>
<td>vsrad</td>
<td>Vector Shift Right Algebraic Doubleword</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4 100003C6</td>
<td>251</td>
<td>V</td>
<td>vcmpbfp[.]</td>
<td>Vector Compare Bounds Single-Precision</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>4 100003C7</td>
<td>234</td>
<td>V</td>
<td>vcmpgtsd[.]</td>
<td>Vector Compare Greater Than Signed Doubleword</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100003CA</td>
<td>247</td>
<td>V</td>
<td>vctxxs</td>
<td>Vector Convert From Single-Precision To Signed Fixed-Point Word Saturate</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100003CE</td>
<td>192</td>
<td>V</td>
<td>vupklpx</td>
<td>Vector Unpack Low Pixel</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 100003D8</td>
<td>679</td>
<td>LMA</td>
<td>mac hlws[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 100003D8</td>
<td>679</td>
<td>LMA</td>
<td>mac hlwsol[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 100003DC</td>
<td>684</td>
<td>LMA</td>
<td>nmach lw[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Saturate Signed</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>XO</td>
<td>4 100003DC</td>
<td>LMA</td>
<td>nmaclhwso[.]</td>
<td>684</td>
<td>Negative Multiply Accumulate Low Halfword to Word Saturate Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000040A</td>
<td>V</td>
<td>vmmaxtp</td>
<td>246</td>
<td>Vector Maximum Single-Precision</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000040B</td>
<td>SP</td>
<td>evmhesmf</td>
<td>608</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000040C</td>
<td>SP</td>
<td>evmhoumi</td>
<td>621</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000040C</td>
<td>V</td>
<td>vslo</td>
<td>200</td>
<td>Vector Shift Left by Octet</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000040D</td>
<td>V</td>
<td>evmhosmi</td>
<td>617</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000040F</td>
<td>SP</td>
<td>evmhosmf</td>
<td>616</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000042B</td>
<td>SP</td>
<td>evmhesmfa</td>
<td>608</td>
<td>Vector Multiply Half Words, Even, Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000042C</td>
<td>SP</td>
<td>evmhoumia</td>
<td>621</td>
<td>Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000042D</td>
<td>SP</td>
<td>evmhosmia</td>
<td>617</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000042F</td>
<td>SP</td>
<td>evmhosmfa</td>
<td>616</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000044A</td>
<td>V</td>
<td>vmminfp</td>
<td>246</td>
<td>Vector Minimum Single-Precision</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000044C</td>
<td>SP</td>
<td>evmwhumi</td>
<td>624</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000044C</td>
<td>V</td>
<td>vsro</td>
<td>201</td>
<td>Vector Shift Right by Octet</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000044D</td>
<td>SP</td>
<td>evmwhsmi</td>
<td>623</td>
<td>Vector Multiply Word High Signed, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 1000044E</td>
<td>V</td>
<td>vpkudum</td>
<td>190</td>
<td>Vector Pack Unsigned Doubleword Unsigned Modulo</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000044F</td>
<td>SP</td>
<td>evmwhsmf</td>
<td>623</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000045B</td>
<td>SP</td>
<td>evmwhsmfa</td>
<td>627</td>
<td>Vector Multiply Word Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000046C</td>
<td>SP</td>
<td>evmwhumia</td>
<td>624</td>
<td>Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000046D</td>
<td>SP</td>
<td>evmwhsmsia</td>
<td>623</td>
<td>Vector Multiply Word High Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000046F</td>
<td>SP</td>
<td>evmwhsmsfa</td>
<td>623</td>
<td>Vector Multiply Word High Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000047B</td>
<td>SP</td>
<td>evmwhsmsfa</td>
<td>627</td>
<td>Vector Multiply Word Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000047C</td>
<td>SP</td>
<td>evaddusiaaw</td>
<td>595</td>
<td>Vector Add Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100004C0</td>
<td>V</td>
<td>vsubudm</td>
<td>210</td>
<td>Vector Subtract Unsigned Doubleword Modulo</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C1</td>
<td>SP</td>
<td>evaddssiaaw</td>
<td>595</td>
<td>Vector Add Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C2</td>
<td>SP</td>
<td>evsubfusiaaw</td>
<td>639</td>
<td>Vector Subtract Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C3</td>
<td>SP</td>
<td>evsubfssiaaw</td>
<td>638</td>
<td>Vector Subtract Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C4</td>
<td>SP</td>
<td>evmra</td>
<td>623</td>
<td>Initialize Accumulator</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100004C4</td>
<td>V</td>
<td>vxor</td>
<td>239</td>
<td>Vector Logical XOR</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C6</td>
<td>SP</td>
<td>evdivws</td>
<td>598</td>
<td>Vector Divide Word Signed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C7</td>
<td>SP</td>
<td>evdivwu</td>
<td>599</td>
<td>Vector Divide Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C8</td>
<td>SP</td>
<td>evaddumiaaw</td>
<td>595</td>
<td>Vector Add Unsigned, Modulo, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4 100004C8</td>
<td>V</td>
<td>vpmsumd</td>
<td>259</td>
<td>Vector Polynomial Multiply-Sum Doubleword</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C9</td>
<td>SP</td>
<td>evaddsmiaaw</td>
<td>594</td>
<td>Vector Add Signed, Modulo, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004CA</td>
<td></td>
<td></td>
<td>639</td>
<td>SP</td>
<td>evsubfumiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004CB</td>
<td></td>
<td></td>
<td>638</td>
<td>SP</td>
<td>evsubfsmiaaw</td>
</tr>
<tr>
<td>VX</td>
<td>4 100004CE</td>
<td></td>
<td></td>
<td>190</td>
<td>V</td>
<td>vpkudus</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000050B</td>
<td></td>
<td></td>
<td>608</td>
<td>SP</td>
<td>evmhesmfaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000050C</td>
<td></td>
<td></td>
<td>622</td>
<td>SP</td>
<td>evmhoumiaaw</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000050C</td>
<td></td>
<td></td>
<td>262</td>
<td>V</td>
<td>vgbbd</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000050D</td>
<td></td>
<td></td>
<td>618</td>
<td>SP</td>
<td>evmhosmiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000050F</td>
<td></td>
<td></td>
<td>617</td>
<td>SP</td>
<td>evmhosmfaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000052B</td>
<td></td>
<td></td>
<td>606</td>
<td>SP</td>
<td>evmhcgsmfwa</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000052C</td>
<td></td>
<td></td>
<td>616</td>
<td>SP</td>
<td>evmhogumiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000052D</td>
<td></td>
<td></td>
<td>615</td>
<td>SP</td>
<td>evmhogsmiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000052F</td>
<td></td>
<td></td>
<td>615</td>
<td>SP</td>
<td>evmhgsmfaaw</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000054C</td>
<td></td>
<td></td>
<td>265</td>
<td>V</td>
<td>vbpermq</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000054C</td>
<td></td>
<td></td>
<td>265</td>
<td>V</td>
<td>vbpermq</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000054D</td>
<td></td>
<td></td>
<td>188</td>
<td>V</td>
<td>vpksdus</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000055B</td>
<td></td>
<td></td>
<td>628</td>
<td>SP</td>
<td>evmwsmfaaa</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000058B</td>
<td></td>
<td></td>
<td>608</td>
<td>SP</td>
<td>evmhesmfanw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000058C</td>
<td></td>
<td></td>
<td>618</td>
<td>SP</td>
<td>evmhoumiaanw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000058D</td>
<td></td>
<td></td>
<td>617</td>
<td>SP</td>
<td>evmhosmianw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000058F</td>
<td></td>
<td></td>
<td>617</td>
<td>SP</td>
<td>evmhosmfanw</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100005A8</td>
<td></td>
<td></td>
<td>607</td>
<td>SP</td>
<td>evmhcgumian</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100005A9</td>
<td></td>
<td></td>
<td>607</td>
<td>SP</td>
<td>evmhcgsmian</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100005AB</td>
<td></td>
<td></td>
<td>606</td>
<td>SP</td>
<td>evmhcgsmfan</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100005AC</td>
<td></td>
<td></td>
<td>616</td>
<td>SP</td>
<td>evmhgumian</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100005AD</td>
<td></td>
<td></td>
<td>615</td>
<td>SP</td>
<td>evmhgsmian</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AF</td>
<td>SP</td>
<td>615</td>
<td></td>
<td>evmhogsfan</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005C0</td>
<td>SP</td>
<td>627</td>
<td></td>
<td>evmlusianw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005C1</td>
<td>SP</td>
<td>625</td>
<td></td>
<td>evmwissianw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100005C4</td>
<td>V</td>
<td>241</td>
<td></td>
<td>vsld</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005C8</td>
<td>SP</td>
<td>626</td>
<td></td>
<td>evmwlusianw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100005CE</td>
<td>V</td>
<td>257</td>
<td></td>
<td>vCrypto vsbox</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005C9</td>
<td>SP</td>
<td>625</td>
<td></td>
<td>evmwlsianw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D3</td>
<td>SP</td>
<td>630</td>
<td></td>
<td>evmwssfan</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D8</td>
<td>SP</td>
<td>631</td>
<td></td>
<td>evmwumian</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005D9</td>
<td>SP</td>
<td>628</td>
<td></td>
<td>evmwmsian</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005DB</td>
<td>SP</td>
<td>628</td>
<td></td>
<td>evmwmsfan</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000064E</td>
<td>V</td>
<td>193</td>
<td></td>
<td>vupkhsw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100006BC</td>
<td>VSX</td>
<td>196</td>
<td></td>
<td>vsrgeow</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100006C2</td>
<td>V.Crypto</td>
<td>258</td>
<td></td>
<td>vsahasigmad</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100006C4</td>
<td>V</td>
<td>242</td>
<td></td>
<td>vsrd</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100006CE</td>
<td>V</td>
<td>193</td>
<td></td>
<td>vupklsw</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100007BC</td>
<td>VSX</td>
<td>196</td>
<td></td>
<td>vmrgew</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100007C2</td>
<td>V</td>
<td>263</td>
<td></td>
<td>vclzd</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100007C3</td>
<td>V</td>
<td>264</td>
<td></td>
<td>vpoppcntld</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>1C000000</td>
<td>B</td>
<td>71</td>
<td></td>
<td>mulli</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>20000000</td>
<td>SR</td>
<td>68</td>
<td></td>
<td>subflic</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>28000000</td>
<td>B</td>
<td>79</td>
<td></td>
<td>cmpli</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>2C000000</td>
<td>B</td>
<td>78</td>
<td></td>
<td>cmpi</td>
</tr>
<tr>
<td>D</td>
<td>12</td>
<td>30000000</td>
<td>SR</td>
<td>67</td>
<td></td>
<td>addic</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>34000000</td>
<td>SR</td>
<td>67</td>
<td></td>
<td>addic.</td>
</tr>
<tr>
<td>D</td>
<td>14</td>
<td>38000000</td>
<td>B</td>
<td>66</td>
<td></td>
<td>addi</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>3C000000</td>
<td>B</td>
<td>66</td>
<td></td>
<td>addis</td>
</tr>
<tr>
<td>B</td>
<td>16</td>
<td>40000000</td>
<td>CT</td>
<td>38</td>
<td></td>
<td>bc[ll][a]</td>
</tr>
<tr>
<td>SC</td>
<td>17</td>
<td>44000000</td>
<td>B</td>
<td>43</td>
<td>863</td>
<td>sc</td>
</tr>
<tr>
<td>I</td>
<td>18</td>
<td>4B000000</td>
<td>B</td>
<td>38</td>
<td></td>
<td>b[ll][a]</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000000</td>
<td>B</td>
<td>42</td>
<td></td>
<td>mcrf</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000020</td>
<td>CT</td>
<td>39</td>
<td></td>
<td>bcrl[]</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000024</td>
<td>P</td>
<td>864</td>
<td></td>
<td>rdfi</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000042</td>
<td>B</td>
<td>42</td>
<td></td>
<td>crnor</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C00004C</td>
<td>P</td>
<td>1042</td>
<td></td>
<td>rfidc</td>
</tr>
<tr>
<td>X</td>
<td>19</td>
<td>4C00004E</td>
<td>P</td>
<td>1042</td>
<td></td>
<td>E.ED rdfi</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000064</td>
<td>P</td>
<td>1041</td>
<td></td>
<td>rfi</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000066</td>
<td>P</td>
<td>1041</td>
<td></td>
<td>rfcfi</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C0000CC</td>
<td>P</td>
<td>1043</td>
<td></td>
<td>E.HV rfgi</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000102</td>
<td>B</td>
<td>42</td>
<td></td>
<td>crandc</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000124</td>
<td>S</td>
<td>820</td>
<td></td>
<td>rfebb</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C00012C</td>
<td>B</td>
<td>776</td>
<td></td>
<td>isync</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Base (hex)</td>
<td>Mode Dep.1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000182</td>
<td>B</td>
<td>41</td>
<td>B</td>
<td>crxor</td>
</tr>
<tr>
<td>XFX</td>
<td>19</td>
<td>4C00018C</td>
<td>E</td>
<td>1228</td>
<td>E</td>
<td>dnh</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C0001C2</td>
<td>B</td>
<td>41</td>
<td>B</td>
<td>crnand</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000202</td>
<td>B</td>
<td>41</td>
<td>B</td>
<td>crand</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000224</td>
<td>H</td>
<td>865</td>
<td>S</td>
<td>hrfd</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000242</td>
<td>B</td>
<td>42</td>
<td>B</td>
<td>creqv</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000324</td>
<td>H</td>
<td>867</td>
<td>S</td>
<td>doze</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000342</td>
<td>B</td>
<td>42</td>
<td>B</td>
<td>croc</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000364</td>
<td>H</td>
<td>867</td>
<td>S</td>
<td>nap</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000382</td>
<td>B</td>
<td>41</td>
<td>B</td>
<td>cor</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C0003A4</td>
<td>H</td>
<td>868</td>
<td>S</td>
<td>sleep</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C0003E4</td>
<td>H</td>
<td>868</td>
<td>S</td>
<td>rwinkle</td>
</tr>
<tr>
<td>XL</td>
<td>19</td>
<td>4C000420</td>
<td>CT</td>
<td>39</td>
<td>B</td>
<td>bcctr[]</td>
</tr>
<tr>
<td>X</td>
<td>19</td>
<td>4C000460</td>
<td>B</td>
<td>40</td>
<td>B</td>
<td>bctar[]</td>
</tr>
<tr>
<td>M</td>
<td>20</td>
<td>50000000</td>
<td>SR</td>
<td>93</td>
<td>B</td>
<td>rlwimi[]</td>
</tr>
<tr>
<td>M</td>
<td>21</td>
<td>54000000</td>
<td>SR</td>
<td>91</td>
<td>B</td>
<td>rlwm[]</td>
</tr>
<tr>
<td>M</td>
<td>23</td>
<td>5C000000</td>
<td>SR</td>
<td>92</td>
<td>B</td>
<td>rlwm[]</td>
</tr>
<tr>
<td>D</td>
<td>24</td>
<td>60000000</td>
<td>B</td>
<td>82</td>
<td>B</td>
<td>ori</td>
</tr>
<tr>
<td>D</td>
<td>25</td>
<td>64000000</td>
<td>B</td>
<td>83</td>
<td>B</td>
<td>oris</td>
</tr>
<tr>
<td>X</td>
<td>26</td>
<td>68000000</td>
<td>B</td>
<td>83</td>
<td>B</td>
<td>xor</td>
</tr>
<tr>
<td>D</td>
<td>27</td>
<td>6C000000</td>
<td>B</td>
<td>83</td>
<td>B</td>
<td>xoris</td>
</tr>
<tr>
<td>D</td>
<td>28</td>
<td>70000000</td>
<td>SR</td>
<td>82</td>
<td>B</td>
<td>andi.</td>
</tr>
<tr>
<td>D</td>
<td>29</td>
<td>74000000</td>
<td>SR</td>
<td>82</td>
<td>B</td>
<td>andis.</td>
</tr>
<tr>
<td>MD</td>
<td>30</td>
<td>78000000</td>
<td>SR</td>
<td>94</td>
<td>64</td>
<td>rldic[]</td>
</tr>
<tr>
<td>MD</td>
<td>30</td>
<td>78000004</td>
<td>SR</td>
<td>94</td>
<td>64</td>
<td>rldicr[]</td>
</tr>
<tr>
<td>MD</td>
<td>30</td>
<td>78000008</td>
<td>SR</td>
<td>95</td>
<td>64</td>
<td>rldic[]</td>
</tr>
<tr>
<td>MDS</td>
<td>30</td>
<td>78000010</td>
<td>SR</td>
<td>95</td>
<td>64</td>
<td>rldic[]</td>
</tr>
<tr>
<td>MDS</td>
<td>30</td>
<td>78000012</td>
<td>SR</td>
<td>96</td>
<td>64</td>
<td>rldicr[]</td>
</tr>
<tr>
<td>MD</td>
<td>30</td>
<td>7800000C</td>
<td>SR</td>
<td>96</td>
<td>64</td>
<td>rldimi[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000000</td>
<td>B</td>
<td>78</td>
<td>B</td>
<td>cmp</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000008</td>
<td>B</td>
<td>80</td>
<td>B</td>
<td>tw</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00000C</td>
<td>V</td>
<td>186</td>
<td>V</td>
<td>lvsl</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00000E</td>
<td>V</td>
<td>184</td>
<td>V</td>
<td>lvebx</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C000010</td>
<td>B</td>
<td>68</td>
<td>B</td>
<td>subfc[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C000012</td>
<td>B</td>
<td>64</td>
<td>64</td>
<td>mulhd[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C000014</td>
<td>B</td>
<td>68</td>
<td>B</td>
<td>addc[]</td>
</tr>
<tr>
<td>XO</td>
<td>31</td>
<td>7C000016</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td>mulhwu[]</td>
</tr>
<tr>
<td>XX1</td>
<td>31</td>
<td>7C000018</td>
<td>VSX</td>
<td>394</td>
<td>VSX</td>
<td>lxsiwzx</td>
</tr>
<tr>
<td>A</td>
<td>31</td>
<td>7C00001E</td>
<td>B</td>
<td>81</td>
<td>B</td>
<td>isel</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000024</td>
<td>E</td>
<td>1134</td>
<td>E</td>
<td>tblibx</td>
</tr>
<tr>
<td>XFX</td>
<td>31</td>
<td>7C000026</td>
<td>B</td>
<td>108</td>
<td>B</td>
<td>mfcr</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000028</td>
<td>B</td>
<td>777</td>
<td>B</td>
<td>lwax</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002A</td>
<td>B</td>
<td>53</td>
<td>64</td>
<td>dx</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002C</td>
<td>E</td>
<td>762</td>
<td>E</td>
<td>icbt</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C00002E</td>
<td>B</td>
<td>51</td>
<td>B</td>
<td>lwzx</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000030</td>
<td>SR</td>
<td>97</td>
<td>B</td>
<td>slw[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000034</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>cntlz[]</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000036</td>
<td>SR</td>
<td>99</td>
<td>64</td>
<td>sid[]</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000038</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>and[.]</td>
<td>AND</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00003A</td>
<td>P</td>
<td>1060</td>
<td>E.PD:64</td>
<td>idex</td>
<td>Load Doubleword by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00003E</td>
<td>P</td>
<td>1060</td>
<td>E.PD</td>
<td>lwepx</td>
<td>Load Word and Zero by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000040</td>
<td>79</td>
<td>B</td>
<td>cmpl</td>
<td>Compare Logical</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00004C</td>
<td>186</td>
<td>V</td>
<td>lvsr</td>
<td>Load Vector for Shift Right</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00004E</td>
<td>181</td>
<td>V</td>
<td>lvehx</td>
<td>Load Vector Element Halfword Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000050</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>subf[.]</td>
<td>Subtract From</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000066</td>
<td>109</td>
<td>VSX</td>
<td>mfsr</td>
<td>Move From VSR Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000068</td>
<td>777</td>
<td>B</td>
<td>lbarx</td>
<td>Load Byte And Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00006A</td>
<td>53</td>
<td>64</td>
<td>ldux</td>
<td>Load Doubleword with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00006C</td>
<td>773</td>
<td>B</td>
<td>dcbst</td>
<td>Data Cache Block Store</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00006E</td>
<td>51</td>
<td>B</td>
<td>lwux</td>
<td>Load Word and Zero with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000074</td>
<td>SR</td>
<td>89</td>
<td>64</td>
<td>cntzd[.]</td>
<td>Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000078</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>andc[.]</td>
<td>AND with Complement</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00007C</td>
<td>791</td>
<td>WT</td>
<td>wait</td>
<td>Wait for Interrupt</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00007E</td>
<td>P</td>
<td>1063</td>
<td>E.PD</td>
<td>dcbstep</td>
<td>Data Cache Block Store by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000088</td>
<td>81</td>
<td>64</td>
<td>td</td>
<td>Trap Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00008E</td>
<td>181</td>
<td>V</td>
<td>lvex</td>
<td>Load Vector Element Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000092</td>
<td>SR</td>
<td>75</td>
<td>64</td>
<td>mulhd[.]</td>
<td>Multiply High Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000094</td>
<td>H</td>
<td>101</td>
<td>BCDA</td>
<td>addg6s</td>
<td>Add and Generate Sixes</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000096</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td>mulhw[.]</td>
<td>Multiply High Word</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000098</td>
<td>393</td>
<td>VSX</td>
<td>lxiwax</td>
<td>Load VSX Scalar as Integer Word Algebraic Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00009C</td>
<td>673</td>
<td>LMM</td>
<td>dlimb[.]</td>
<td>Determine Leftmost Zero Byte</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000A6</td>
<td>P</td>
<td>888</td>
<td>1055</td>
<td>S</td>
<td>mfmsr</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000A8</td>
<td>782</td>
<td>64</td>
<td>ldax</td>
<td>Load Doubleword And Reserve Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000AC</td>
<td>773</td>
<td>B</td>
<td>dcbf</td>
<td>Data Cache Block Flush</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000AE</td>
<td>49</td>
<td>B</td>
<td>lbox</td>
<td>Load Byte and Zero Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000BE</td>
<td>P</td>
<td>1059</td>
<td>E.PD</td>
<td>lbpex</td>
<td>Load Byte and Zero by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000CE</td>
<td>182</td>
<td>V</td>
<td>lvx</td>
<td>Load Vector Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0000D0</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>neg[.]</td>
<td>Negate</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C0000E6</td>
<td>109</td>
<td>VSX</td>
<td>mfsrwx</td>
<td>Move From VSR Word and Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000E8</td>
<td>778</td>
<td>B</td>
<td>lhax</td>
<td>Load Halfword And Reserve Indexed Xform</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000EE</td>
<td>48</td>
<td>B</td>
<td>lbux</td>
<td>Load Byte and Zero with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000F4</td>
<td>87</td>
<td>B</td>
<td>popcntb</td>
<td>Population Count Byte-wise</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000F8</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>nor[.]</td>
<td>NOR</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000FE</td>
<td>P</td>
<td>1064</td>
<td>E.PD</td>
<td>dcbfep</td>
<td>Data Cache Block Flush by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000106</td>
<td>P</td>
<td>1056</td>
<td>E</td>
<td>wrtee</td>
<td>Write External Enable</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00010C</td>
<td>M</td>
<td>1122</td>
<td>ECL</td>
<td>dcbstsl</td>
<td>Data Cache Block Touch for Store and Lock Set</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00010E</td>
<td>184</td>
<td>V</td>
<td>stvex</td>
<td>Store Vector Element Byte Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000110</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfe[.]</td>
<td>Subtract From Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000114</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>addfe[.]</td>
<td>Add Extended</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000118</td>
<td>394</td>
<td>VSX</td>
<td>sbxswx</td>
<td>Store VSX Scalar as Integer Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00011C</td>
<td>P</td>
<td>1007</td>
<td>S.PC</td>
<td>msgsndp</td>
<td>Message Send Privileged</td>
</tr>
<tr>
<td>XFX</td>
<td>31 7C000120</td>
<td>107</td>
<td>B</td>
<td>mtcfr</td>
<td>Move To Condition Register Fields</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000124</td>
<td>P</td>
<td>884</td>
<td>1055</td>
<td>E</td>
<td>mtmsr</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000126</td>
<td>108</td>
<td>S</td>
<td>mtsle</td>
<td>Move To Split Little Endian</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00012A</td>
<td>57</td>
<td>64</td>
<td>stdx</td>
<td>Store Doubleword Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00012D</td>
<td>781</td>
<td>B</td>
<td>stwx</td>
<td>Store Word Conditional Indexed &amp; record CR0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00012E</td>
<td>56</td>
<td>B</td>
<td>stwx</td>
<td>Store Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000134</td>
<td>88</td>
<td>B</td>
<td>prtyw</td>
<td>Parity Word</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00013A</td>
<td>P 1062</td>
<td>E.PD:64</td>
<td>stdex</td>
<td>E.PD</td>
<td>Store Doubleword by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00013E</td>
<td>P 1062</td>
<td>E.PD</td>
<td>stwex</td>
<td>E.PD</td>
<td>Store Word by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000146</td>
<td>P 1057</td>
<td>E</td>
<td>wteei</td>
<td>E.PD</td>
<td>Write External Enable Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00014C</td>
<td>M 1122</td>
<td>ECL</td>
<td>dcbtls</td>
<td>E.PD</td>
<td>Data Cache Block Touch and Lock Set</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00014E</td>
<td>V 184</td>
<td>V</td>
<td>stvehx</td>
<td>E.PD</td>
<td>Store Vector Element Halfword Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00015C</td>
<td>P 1007</td>
<td>S.PC</td>
<td>msgclrp</td>
<td>E.PD</td>
<td>Message Clear Privileged</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000164</td>
<td>P 886</td>
<td>S</td>
<td>mtnsrd</td>
<td>E.PD</td>
<td>Move To Machine State Register Doubleword</td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C000166</td>
<td>110</td>
<td>VSX</td>
<td>mtsrdr</td>
<td>VSX</td>
<td>Move To VSR Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00016A</td>
<td>57</td>
<td>64</td>
<td>stdux</td>
<td>E.PD</td>
<td>Store Doubleword with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00016D</td>
<td>785</td>
<td>LSQ</td>
<td>stqcx.</td>
<td>VSX</td>
<td>Store Quadword Conditional Indexed and record CR0</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00016E</td>
<td>56</td>
<td>B</td>
<td>stwux</td>
<td>S.PC</td>
<td>Store Word with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000174</td>
<td>88</td>
<td>64</td>
<td>prtyd</td>
<td>E.PD</td>
<td>Parity Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00018D</td>
<td>1121</td>
<td>ECL</td>
<td>icblq.</td>
<td>E.PD</td>
<td>Instruction Cache Block Lock Query</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00018E</td>
<td>V 185</td>
<td>V</td>
<td>stvewx</td>
<td>E.PD</td>
<td>Store Vector Element Word Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000190</td>
<td>SR 70</td>
<td>B</td>
<td>subfze[.]</td>
<td>E.PD</td>
<td>Subtract From Zero Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000194</td>
<td>SR 70</td>
<td>B</td>
<td>addze[.]</td>
<td>E.PD</td>
<td>Add to Zero Extended</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00019C</td>
<td>H 1006</td>
<td>S.PC</td>
<td>msgsnd</td>
<td>E.PD</td>
<td>Message Send</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001A4</td>
<td>P 926</td>
<td>S</td>
<td>mtsr</td>
<td>VSX</td>
<td>Move To Segment Register</td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C0001A6</td>
<td>110</td>
<td>VSX</td>
<td>mtvswa</td>
<td>VSX</td>
<td>Move To VSR Word Algebraic</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001AD</td>
<td>782</td>
<td>64</td>
<td>stdcx.</td>
<td>E.PD</td>
<td>Store Doubleword Conditional Indexed &amp; record CR0</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001AE</td>
<td>54</td>
<td>B</td>
<td>stbx</td>
<td>E.PD</td>
<td>Store Byte Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001BE</td>
<td>P 1061</td>
<td>E.PD</td>
<td>stbex</td>
<td>E.PD</td>
<td>Store Byte by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001CC</td>
<td>M 1124</td>
<td>ECL</td>
<td>icbcl</td>
<td>E.PD</td>
<td>Instruction Cache Block Lock Clear</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001CE</td>
<td>V 182</td>
<td>V</td>
<td>stvx</td>
<td>E.PD</td>
<td>Store Vector Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D0</td>
<td>SR 69</td>
<td>B</td>
<td>subfme[.]</td>
<td>E.PD</td>
<td>Subtract From Minus One Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D2</td>
<td>SR 64</td>
<td>64</td>
<td>mulld[.]</td>
<td>E.PD</td>
<td>Multiply Low Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D4</td>
<td>SR 69</td>
<td>B</td>
<td>addme[.]</td>
<td>E.PD</td>
<td>Add to Minus One Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D6</td>
<td>SR 71</td>
<td>B</td>
<td>muliw[.]</td>
<td>E.PD</td>
<td>Multiply Low Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001DC</td>
<td>H 1006</td>
<td>S.PC</td>
<td>msgscl</td>
<td>E.PD</td>
<td>Message Clear</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001E4</td>
<td>P 926</td>
<td>S</td>
<td>mtsrin</td>
<td>VSX</td>
<td>Move To Segment Register Indirect</td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C0001E6</td>
<td>111</td>
<td>VSX</td>
<td>mtvswz</td>
<td>VSX</td>
<td>Move To VSR Word and Zero</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001EC</td>
<td>771</td>
<td>B</td>
<td>dbtist</td>
<td>E.PD</td>
<td>Data Cache Block Touch for Store</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001EE</td>
<td>54</td>
<td>B</td>
<td>stbux</td>
<td>E.PD</td>
<td>Store Byte with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001EF</td>
<td>90</td>
<td>64</td>
<td>bpermidd</td>
<td>E.PD</td>
<td>Bit Permute Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001FE</td>
<td>P 1066</td>
<td>E.PD</td>
<td>dbtbstep</td>
<td>E.PD</td>
<td>Data Cache Block Touch for Store by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000206</td>
<td>P 1055</td>
<td>E.DC</td>
<td>mfdcrx</td>
<td>E.PD</td>
<td>Move From Device Control Register Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00020E</td>
<td>P 1070</td>
<td>E.PD</td>
<td>mvex</td>
<td>E.PD</td>
<td>Move From Device Control Register Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000214</td>
<td>SR 67</td>
<td>B</td>
<td>add[.]</td>
<td>E.PD</td>
<td>Add</td>
</tr>
<tr>
<td>XL</td>
<td>31 7C00021C</td>
<td>P 1043</td>
<td>E.HV</td>
<td>ehpri</td>
<td>E.PD</td>
<td>Embedded Hypervisor Privilege</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000224</td>
<td>P 931</td>
<td>S</td>
<td>tlbil</td>
<td>E.PD</td>
<td>TLB Invalidate Entry Local</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00022B</td>
<td>784</td>
<td>LSQ</td>
<td>tgrabx</td>
<td>E.PD</td>
<td>Load Quadword And Reserve Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00022C</td>
<td>770</td>
<td>B</td>
<td>dbct</td>
<td>E.PD</td>
<td>Data Cache Block Touch</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00022E</td>
<td>49</td>
<td>B</td>
<td>lhzx</td>
<td>E.PD</td>
<td>Load Halfword and Zero Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000234</td>
<td>H 101</td>
<td>BCDCA</td>
<td>cdttbd</td>
<td>E.PD</td>
<td>Convert Declets To Binary Coded Decimal</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000238</td>
<td>SR 85</td>
<td>B</td>
<td>eqv[.]</td>
<td>E.PD</td>
<td>Equivalent</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00023E</td>
<td>P 1059</td>
<td>E.PD</td>
<td>lhepx</td>
<td>E.PD</td>
<td>Load Halfword and Zero by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000246</td>
<td>113</td>
<td>E.DC</td>
<td>mfdcrux</td>
<td>E.PD</td>
<td>Move From Device Control Register User Mode Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00024E</td>
<td>P 1070</td>
<td>E.PD</td>
<td>lvepx</td>
<td>E.PD</td>
<td>Load Vector by External PID Indexed</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO0025C</td>
<td>44</td>
<td>S</td>
<td>mfbhrbe</td>
<td>Move From Branch History Rolling Buffer</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00264</td>
<td>64</td>
<td>H</td>
<td>928</td>
<td>S</td>
<td>lbie</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0026C</td>
<td>826</td>
<td>EC</td>
<td>eciwx</td>
<td>External Control In Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0026E</td>
<td>49</td>
<td>B</td>
<td>lhzux</td>
<td>Load Halfword and Zero with Updated Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00274</td>
<td>H</td>
<td>101</td>
<td>BCDA</td>
<td>cbcdld</td>
<td>Convert Binary Coded Decimal To Declets</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00278</td>
<td>84</td>
<td>B</td>
<td>xor[.]</td>
<td>XOR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0027E</td>
<td>P</td>
<td>1063</td>
<td>E.PD</td>
<td>dcbtep</td>
<td>Data Cache Block Touch by External PID</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO0026E</td>
<td>P</td>
<td>1055</td>
<td>E.DC</td>
<td>mfdr</td>
<td>Move From Device Control Register</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0028C</td>
<td>P</td>
<td>1242</td>
<td>E.CD</td>
<td>dcread</td>
<td>Data Cache Read</td>
</tr>
<tr>
<td>XX1</td>
<td>31 TCO0029B</td>
<td>395</td>
<td>VSX</td>
<td>lxvdsx</td>
<td>Load VSR Vector Doubleword &amp; Splat Indexed</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO0029C</td>
<td>O</td>
<td>1257</td>
<td>E.PM</td>
<td>mtpmr</td>
<td>Move from Performance Monitor Register</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO002A6</td>
<td>O</td>
<td>106</td>
<td>B</td>
<td>mfspr</td>
<td>Move From Special Purpose Register</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002AA</td>
<td>52</td>
<td>64</td>
<td>lwax</td>
<td>Load Word Algebraic Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002AE</td>
<td>50</td>
<td>B</td>
<td>lhax</td>
<td>Load Halfword Algebraic Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002CE</td>
<td>182</td>
<td>V</td>
<td>lvxl</td>
<td>Load Vector Indexed Last</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002E4</td>
<td>H</td>
<td>933</td>
<td>S</td>
<td>tibia</td>
<td>TLB Invalidate All</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO002E6</td>
<td>813</td>
<td>S.out</td>
<td>mftb</td>
<td>Move From Time Base</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002EA</td>
<td>52</td>
<td>64</td>
<td>lwaux</td>
<td>Load Word Algebraic with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002EE</td>
<td>50</td>
<td>B</td>
<td>lhaux</td>
<td>Load Halfword Algebraic with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO002F4</td>
<td>87</td>
<td>B</td>
<td>popcntw</td>
<td>Population Count Words</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00306</td>
<td>P</td>
<td>1054</td>
<td>E.DC</td>
<td>mtdcr</td>
<td>Move To Device Control Register Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0030C</td>
<td>M</td>
<td>1123</td>
<td>ECL</td>
<td>dcblc</td>
<td>Data Cache Block Lock Clear</td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO0032L</td>
<td>SR</td>
<td>77</td>
<td>64</td>
<td>divdeu[.]</td>
<td>Divide Doubleword Extended Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO0031E</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>diweu[.]</td>
<td>Divide Word Extended Unsigned</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00324</td>
<td>P</td>
<td>921</td>
<td>S</td>
<td>sbmte</td>
<td>SLB Move To Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0032E</td>
<td>55</td>
<td>B</td>
<td>sthx</td>
<td>Store Halfword Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0033E</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>orc[.]</td>
<td>OR with Complement</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0033E</td>
<td>P</td>
<td>1061</td>
<td>E.PD</td>
<td>sthepx</td>
<td>Store Halfword by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00346</td>
<td>113</td>
<td>E.DC</td>
<td>mtdcru</td>
<td>Move To Device Control Register User Mode Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO0035D</td>
<td>1121</td>
<td>ECL</td>
<td>dcblq,</td>
<td>Data Cache Block Lock Query</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO00352</td>
<td>SR</td>
<td>77</td>
<td>64</td>
<td>divde[.]</td>
<td>Divide Doubleword Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO00356</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>diwe[.]</td>
<td>Divide Word Extended</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0035C</td>
<td>44</td>
<td>S</td>
<td>crbbrb</td>
<td>Clear BHRB</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00364</td>
<td>P</td>
<td>919</td>
<td>S</td>
<td>sbie</td>
<td>SLB Invalidate Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0036C</td>
<td>826</td>
<td>EC</td>
<td>ecowx</td>
<td>External Control Out Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0036E</td>
<td>55</td>
<td>B</td>
<td>sthux</td>
<td>Store Halfword with Update Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 TCO00378</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>or[.]</td>
<td>OR</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO00386</td>
<td>P</td>
<td>1054</td>
<td>E.DC</td>
<td>mthux</td>
<td>Move To Device Control Register</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO0038C</td>
<td>P</td>
<td>1239</td>
<td>E.CI</td>
<td>dci</td>
<td>Data Cache Invalidate</td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO00392</td>
<td>SR</td>
<td>76</td>
<td>64</td>
<td>divdu[.]</td>
<td>Divide Doubleword Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>31 TCO00396</td>
<td>SR</td>
<td>72</td>
<td>B</td>
<td>diwu[.]</td>
<td>Divide Word Unsigned</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO0039C</td>
<td>O</td>
<td>1257</td>
<td>E.PM</td>
<td>mtpmr</td>
<td>Move To Performance Monitor Register</td>
</tr>
<tr>
<td>XFX</td>
<td>31 TCO003A6</td>
<td>O</td>
<td>104</td>
<td>B</td>
<td>mtspr</td>
<td>Move To Special Purpose Register</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO003AC</td>
<td>P</td>
<td>1118</td>
<td>E</td>
<td>dcbi</td>
<td>Data Cache Block Invalidate</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO003BB</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td>nand[.]</td>
<td>NAND</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO003C6</td>
<td>P</td>
<td>824</td>
<td>DS</td>
<td>dsn</td>
<td>Decorated Storage Notify</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO003CC</td>
<td>P</td>
<td>1242</td>
<td>E.CD</td>
<td>dcread</td>
<td>Data Cache Read</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO003CC</td>
<td>M</td>
<td>1123</td>
<td>ECL</td>
<td>icbts</td>
<td>Instruction Cache Block Touch and Lock Set</td>
</tr>
<tr>
<td>X</td>
<td>31 TCO003CE</td>
<td>185</td>
<td>V</td>
<td>stvxl</td>
<td>Store Vector Indexed Last</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Base (hex)</td>
<td>Mode Dep. ^2</td>
<td>Privilege ^1</td>
<td>Page</td>
<td>Category ^1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>---------------</td>
<td>---------------</td>
<td>------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0003D2</td>
<td>SR</td>
<td>76</td>
<td>64</td>
<td>divd[.]</td>
<td>Divide Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0003D6</td>
<td>SR</td>
<td>72</td>
<td>B</td>
<td>divw[.]</td>
<td>Divide Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003E4</td>
<td>P</td>
<td>920</td>
<td>S</td>
<td>slbia</td>
<td>SLB Invalidate All</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003F4</td>
<td>P</td>
<td>89</td>
<td>64</td>
<td>popcntd</td>
<td>Population Count Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003F8</td>
<td>E</td>
<td>86</td>
<td>B</td>
<td>cmpb</td>
<td>Compare Byte</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000400</td>
<td></td>
<td>113</td>
<td>E</td>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000406</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>lbdx</td>
<td>Load Byte with Decoration Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000410</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>subfc[.]</td>
<td>Subtract From &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000414</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>addc[.]</td>
<td>Add Carrying &amp; record OV</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000418</td>
<td></td>
<td>394</td>
<td></td>
<td>lxspx</td>
<td>Load VSX Scalar Single-Precision Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000428</td>
<td></td>
<td>60</td>
<td>64</td>
<td>ldbx</td>
<td>Load Doubleword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00042A</td>
<td>MA</td>
<td>63</td>
<td></td>
<td>lsdx</td>
<td>Load String Word Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00042C</td>
<td>MA</td>
<td>59</td>
<td></td>
<td>lwbx</td>
<td>Load Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00042E</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>lfsx</td>
<td>Load Floating-Point Single Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000430</td>
<td>SR</td>
<td>97</td>
<td></td>
<td>sr[.]</td>
<td>Shift Right Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000436</td>
<td>SR</td>
<td>99</td>
<td>64</td>
<td>srdf[.]</td>
<td>Shift Right Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000446</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>lhdx</td>
<td>Load Halfword with Decoration Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000450</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>subfo[.]</td>
<td>Subtract From &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00046C</td>
<td>H</td>
<td>934</td>
<td>141</td>
<td>tblsync</td>
<td>TLB Synchronize</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00046E</td>
<td>P</td>
<td>138</td>
<td></td>
<td>lfsux</td>
<td>Load Floating-Point Single with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000486</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>lwdx</td>
<td>Load Word with Decoration Indexed</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000498</td>
<td>VSX</td>
<td>393</td>
<td></td>
<td>lxsdx</td>
<td>Load VSR Scalar Doubleword Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004A6</td>
<td>32 P</td>
<td>927</td>
<td>S</td>
<td>mfsr</td>
<td>Move From Segment Register</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004AA</td>
<td>MA</td>
<td>63</td>
<td></td>
<td>lswi</td>
<td>Load String Word Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004AC</td>
<td>B</td>
<td>786</td>
<td></td>
<td>sync</td>
<td>Synchronize</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004AE</td>
<td>FP</td>
<td>135</td>
<td></td>
<td>lfdx</td>
<td>Load Floating-Point Double Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004BE</td>
<td>P</td>
<td>1068</td>
<td>E.PD</td>
<td>lfdexpx</td>
<td>Load Floating-Point Double by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004C6</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>lddx</td>
<td>Load Doubleword with Decoration Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0004D0</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>nego[.]</td>
<td>Negate &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004EE</td>
<td>FP</td>
<td>135</td>
<td></td>
<td>lfdux</td>
<td>Load Floating-Point Double with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000506</td>
<td>DS</td>
<td>823</td>
<td></td>
<td>stdbx</td>
<td>Store Byte with Decoration Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000510</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfe[.]</td>
<td>Subtract From Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000514</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>adeo[.]</td>
<td>Add Extended &amp; record OV &amp; record OV</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000518</td>
<td>VSX</td>
<td>394</td>
<td></td>
<td>stxspx</td>
<td>Store VSR Scalar Word Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00051D</td>
<td>TM</td>
<td>806</td>
<td></td>
<td>tbegin.</td>
<td>Transaction Begin</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000526</td>
<td>32 P</td>
<td>927</td>
<td>S</td>
<td>mfsrin</td>
<td>Move From Segment Register Indirect</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000528</td>
<td>MA</td>
<td>64</td>
<td></td>
<td>stdbrx</td>
<td>Store Doubleword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00052A</td>
<td>MA</td>
<td>64</td>
<td></td>
<td>stswx</td>
<td>Store String Word Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00052C</td>
<td>B</td>
<td>59</td>
<td></td>
<td>stwbrx</td>
<td>Store Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00052E</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>stfsx</td>
<td>Store Floating-Point Single Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000546</td>
<td>DS</td>
<td>823</td>
<td></td>
<td>sthdx</td>
<td>Store Halfword with Decoration Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00055C</td>
<td>TM</td>
<td>807</td>
<td></td>
<td>tend.</td>
<td>Transaction End</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000562</td>
<td>B</td>
<td>779</td>
<td></td>
<td>stdbox.</td>
<td>Store Byte Conditional Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000566</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>stfsux</td>
<td>Store Floating-Point Single with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000586</td>
<td>DS</td>
<td>823</td>
<td></td>
<td>stwdx</td>
<td>Store Word with Decoration Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000590</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>subfzeo[.]</td>
<td>Subtract From Zero Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000594</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>addzeo[.]</td>
<td>Add to Zero Extended &amp; record OV</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000598</td>
<td>VSX</td>
<td>398</td>
<td></td>
<td>stxsspx</td>
<td>Store VSR Scalar Doubleword Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00059C</td>
<td>TM</td>
<td>811</td>
<td></td>
<td>tcheck</td>
<td>Transaction Check</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005AA</td>
<td>MA</td>
<td>64</td>
<td></td>
<td>stswi</td>
<td>Store String Word Immediate</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005AD</td>
<td>B</td>
<td>stthcx.</td>
<td>780</td>
<td>DS</td>
<td>Store Halfword Conditional Indexed Xform</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005AE</td>
<td>FP</td>
<td>stfdx</td>
<td>139</td>
<td>FP</td>
<td>Store Floating-Point Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005BE</td>
<td>E.PD</td>
<td>stfdepz</td>
<td>1068</td>
<td>E.PD</td>
<td>Store Floating-Point Double by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005C6</td>
<td>DS</td>
<td>stddx</td>
<td>823</td>
<td>DS</td>
<td>Store Doubleword with Decoration Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005D0</td>
<td>B</td>
<td>subfneo[.]</td>
<td>69</td>
<td>B</td>
<td>Subtract From Minus One Extended &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005D2</td>
<td>64</td>
<td>mulldo[.]</td>
<td>64</td>
<td>64</td>
<td>Multiply Low Doubleword &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005D4</td>
<td>B</td>
<td>addmeo[.]</td>
<td>69</td>
<td>B</td>
<td>Add to Minus One Extended &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005D6</td>
<td>71</td>
<td>mulwo[.]</td>
<td>71</td>
<td>B</td>
<td>Multiply Low Word &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005D8</td>
<td>E</td>
<td>lcbia</td>
<td>770</td>
<td>E</td>
<td>Data Cache Block Allocate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005EE</td>
<td>E.PD</td>
<td>stfdx</td>
<td>139</td>
<td>E.PD</td>
<td>Store Floating-Point Double with Update Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00060E</td>
<td>E.PD</td>
<td>stvepxl</td>
<td>1071</td>
<td>E.PD</td>
<td>Store Vector by External PID Indexed Last</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000614</td>
<td>67</td>
<td>addo[.]</td>
<td>67</td>
<td>B</td>
<td>Add &amp; record OV</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000618</td>
<td>VSX</td>
<td>lxvw4x</td>
<td>396</td>
<td>VSX</td>
<td>Load VSR Vector Word*4 Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00061D</td>
<td>TM</td>
<td>tabortwc.</td>
<td>809</td>
<td>TM</td>
<td>Transaction Abort Word Conditional</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000624</td>
<td>P</td>
<td>tblivax</td>
<td>1132</td>
<td>E</td>
<td>TLB Invalidate Virtual Address Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000626</td>
<td>H</td>
<td>lwzci</td>
<td>876</td>
<td>S</td>
<td>Load Word and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00062C</td>
<td>B</td>
<td>lhbrx</td>
<td>59</td>
<td>B</td>
<td>Load Halfword Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00062E</td>
<td>FP.out</td>
<td>lfdpx</td>
<td>141</td>
<td>FP</td>
<td>Load Floating-Point Double Pair Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000630</td>
<td>SR</td>
<td>sraw[.]</td>
<td>98</td>
<td>B</td>
<td>Shift Right Algebraic Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000634</td>
<td>SR</td>
<td>sradi[.]</td>
<td>100</td>
<td>64</td>
<td>Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>31 7C00063E</td>
<td>P</td>
<td>evlddepx</td>
<td>1069</td>
<td>E.PD</td>
<td>Vector Load Double Word into Double Word by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000646</td>
<td>DS</td>
<td>lfdx</td>
<td>822</td>
<td>DS</td>
<td>Load Floating Doubleword with Decoration Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00064E</td>
<td>P</td>
<td>stvepx</td>
<td>1071</td>
<td>E.PD</td>
<td>Store Vector by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00065D</td>
<td>TM</td>
<td>tabortdc.</td>
<td>809</td>
<td>TM</td>
<td>Transaction Abort Doubleword Conditional</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00066A</td>
<td>H</td>
<td>lhzcix</td>
<td>876</td>
<td>S</td>
<td>Load Halfword and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000670</td>
<td>SR</td>
<td>sraw[.]</td>
<td>98</td>
<td>B</td>
<td>Shift Right Algebraic Word Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000674</td>
<td>SR</td>
<td>sradi[.]</td>
<td>100</td>
<td>64</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000698</td>
<td>VSX</td>
<td>lxvd2x</td>
<td>395</td>
<td>VSX</td>
<td>Load VSR Vector Doubleword*2 Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00069D</td>
<td>TM</td>
<td>tabortwci.</td>
<td>809</td>
<td>TM</td>
<td>Transaction Abort Word Conditional Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006A5</td>
<td>P</td>
<td>tblsrx.</td>
<td>1138</td>
<td>E.TWC</td>
<td>TLB Search and Reserve Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006A6</td>
<td>P</td>
<td>slbmfiev</td>
<td>922</td>
<td>S</td>
<td>SLB Move From Entry VSID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006AA</td>
<td>H</td>
<td>lbzcix</td>
<td>876</td>
<td>S</td>
<td>Load Byte and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006AC</td>
<td>S</td>
<td>eieio</td>
<td>790</td>
<td>S</td>
<td>Enforce In-order Execution of I/O</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006AC</td>
<td>E</td>
<td>mbar</td>
<td>790</td>
<td>E</td>
<td>Memory Barrier</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006AE</td>
<td>FP</td>
<td>lfiwax</td>
<td>136</td>
<td>FP</td>
<td>Load Floating-Point as Integer Word Algebraic Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006DD</td>
<td>TM</td>
<td>tabortdc.</td>
<td>810</td>
<td>TM</td>
<td>Transaction Abort Doubleword Conditional Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006EA</td>
<td>H</td>
<td>lddcix</td>
<td>876</td>
<td>S</td>
<td>Load Doubleword Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006EE</td>
<td>FP</td>
<td>lfiwzx</td>
<td>136</td>
<td>FP</td>
<td>Load Floating-Point as Integer Word and Zero Indexed</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000712</td>
<td>SR</td>
<td>divideu0[.]</td>
<td>77</td>
<td>64</td>
<td>Divide Doubleword Extended Unsigned &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000716</td>
<td>B</td>
<td>divwe0[.]</td>
<td>73</td>
<td>B</td>
<td>Divide Word Extended Unsigned &amp; record OV</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000718</td>
<td>VSX</td>
<td>sbxwc4x</td>
<td>398</td>
<td>VSX</td>
<td>Store VSR Vector Word*4 Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00071D</td>
<td>TM</td>
<td>tabort.</td>
<td>808</td>
<td>TM</td>
<td>Transaction Abort</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000724</td>
<td>P</td>
<td>1136</td>
<td>E</td>
<td>tblbx</td>
<td>TLB Search Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000726</td>
<td>P</td>
<td>923</td>
<td>S</td>
<td>slbfmee</td>
<td>SLB Move From Entry ESID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00072A</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stwcix</td>
<td>Store Word and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00072C</td>
<td>59</td>
<td>B</td>
<td>stbrx</td>
<td>Store Halfword Byte-Reverse Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00072E</td>
<td>141</td>
<td>FP.out</td>
<td>stfdpx</td>
<td>Store Floating-Point Double Pair Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000734</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>extsh[.]</td>
<td>Extend Sign Halfword</td>
</tr>
<tr>
<td>EVX</td>
<td>31 7C00073E</td>
<td>P</td>
<td>1069</td>
<td>E.PD</td>
<td>evstddep[x]</td>
<td>Vector Store Double of Double by External PID Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000746</td>
<td>823</td>
<td>DS</td>
<td>stfdx</td>
<td>Store Floating Doubleword with Decoration Indexed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000752</td>
<td>SR</td>
<td>77</td>
<td>64</td>
<td>divdeo[.]</td>
<td>Divide Doubleword Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000756</td>
<td>SR</td>
<td>73</td>
<td>B</td>
<td>divwo[.]</td>
<td>Divide Word Extended &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00075D</td>
<td>879</td>
<td>TM</td>
<td>treclaim.</td>
<td>Transaction Reclaim</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000764</td>
<td>P</td>
<td>1139</td>
<td>E</td>
<td>tlbre</td>
<td>TLB Read Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00076A</td>
<td>SR</td>
<td>877</td>
<td>S</td>
<td>sthcix</td>
<td>Store Halfword and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000774</td>
<td>SR</td>
<td>85</td>
<td>B</td>
<td>extsb[.]</td>
<td>Extend Sign Byte</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000778</td>
<td>P</td>
<td>1239</td>
<td>E.Cl</td>
<td>ici</td>
<td>Instruction Cache Invalidate</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000792</td>
<td>SR</td>
<td>76</td>
<td>64</td>
<td>divdvo[.]</td>
<td>Divide Doubleword Unsigned &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000796</td>
<td>SR</td>
<td>72</td>
<td>B</td>
<td>diwuo[.]</td>
<td>Divide Word Unsigned &amp; record OV</td>
</tr>
<tr>
<td>XX1</td>
<td>31 7C000798</td>
<td>398</td>
<td>VSX</td>
<td>stxvd2x</td>
<td>Store VSR Vector Doubleword*2 Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007A4</td>
<td>P</td>
<td>1141</td>
<td>E</td>
<td>tblwe</td>
<td>TLB Write Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007A7</td>
<td>P</td>
<td>923</td>
<td>S</td>
<td>slbfmee</td>
<td>SLB Find Entry ESID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007AA</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stbcix</td>
<td>Store Byte Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007AC</td>
<td>762</td>
<td>B</td>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007AE</td>
<td>140</td>
<td>FP</td>
<td>stfiwx</td>
<td>Store Floating-Point as Integer Word Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007B4</td>
<td>SR</td>
<td>89</td>
<td>64</td>
<td>extsw[.]</td>
<td>Extend Sign Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007BE</td>
<td>P</td>
<td>1067</td>
<td>E.PD</td>
<td>icbiep</td>
<td>Instruction Cache Block Invalidate by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007CC</td>
<td>P</td>
<td>1243</td>
<td>E.CD</td>
<td>icread</td>
<td>Instruction Cache Read</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0007D2</td>
<td>SR</td>
<td>76</td>
<td>64</td>
<td>divdvo[.]</td>
<td>Divide Doubleword &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0007D6</td>
<td>SR</td>
<td>72</td>
<td>B</td>
<td>diwuo[.]</td>
<td>Divide Word &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007DD</td>
<td>880</td>
<td>TM</td>
<td>trechkpt.</td>
<td>Transaction Recheckpoint</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007EA</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stdcix</td>
<td>Store Doubleword Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007EC</td>
<td>773</td>
<td>B</td>
<td>dcbz</td>
<td>Data Cache Block Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007FE</td>
<td>P</td>
<td>1067</td>
<td>E.PD</td>
<td>dcbzep</td>
<td>Data Cache Block Zero by External PID</td>
</tr>
<tr>
<td>XFX</td>
<td>31 7C100026</td>
<td>112</td>
<td>B</td>
<td>mfcorf</td>
<td>Move From One Condition Register Field</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>31 7C100120</td>
<td>112</td>
<td>B</td>
<td>mtcorf</td>
<td>Move To One Condition Register Field</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>32 80000000</td>
<td>51</td>
<td>B</td>
<td>lwz</td>
<td>Load Word and Zero</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>33 84000000</td>
<td>51</td>
<td>B</td>
<td>lwzu</td>
<td>Load Word and Zero with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>34 88000000</td>
<td>48</td>
<td>B</td>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>35 8C000000</td>
<td>48</td>
<td>B</td>
<td>lbzu</td>
<td>Load Byte and Zero with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>36 90000000</td>
<td>56</td>
<td>B</td>
<td>stw</td>
<td>Store Word</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>37 94000000</td>
<td>56</td>
<td>B</td>
<td>stwu</td>
<td>Store Word with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>38 98000000</td>
<td>54</td>
<td>B</td>
<td>stb</td>
<td>Store Byte</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>39 9C000000</td>
<td>54</td>
<td>B</td>
<td>stbu</td>
<td>Store Byte with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>40 A0000000</td>
<td>49</td>
<td>B</td>
<td>lhz</td>
<td>Load Halfword and Zero</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>41 A4000000</td>
<td>49</td>
<td>B</td>
<td>lhzu</td>
<td>Load Halfword and Zero with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>42 A8000000</td>
<td>50</td>
<td>B</td>
<td>lha</td>
<td>Load Halfword Algebraic</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>43 AC000000</td>
<td>50</td>
<td>B</td>
<td>lhau</td>
<td>Load Halfword Algebraic with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>44 B0000000</td>
<td>55</td>
<td>B</td>
<td>sth</td>
<td>Store Halfword</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>45 B4000000</td>
<td>55</td>
<td>B</td>
<td>sthu</td>
<td>Store Halfword with Update</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>46 B8000000</td>
<td>61</td>
<td>B</td>
<td>lmw</td>
<td>Load Multiple Word</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>47 BC000000</td>
<td>61</td>
<td>B</td>
<td>stmw</td>
<td>Store Multiple Word</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>D</td>
<td>48 C0000000</td>
<td>138 FP</td>
<td>lfs</td>
<td>Load Floating-Point Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>49 C4000000</td>
<td>138 FP</td>
<td>lfsu</td>
<td>Load Floating-Point Single with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>50 C8000000</td>
<td>135 FP</td>
<td>lfd</td>
<td>Load Floating-Point Double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>51 CC000000</td>
<td>135 FP</td>
<td>lfdu</td>
<td>Load Floating-Point Double with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>52 D0000000</td>
<td>138 FP</td>
<td>stfs</td>
<td>Store Floating-Point Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>53 D4000000</td>
<td>138 FP</td>
<td>stfsu</td>
<td>Store Floating-Point Single with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>54 D8000000</td>
<td>139 FP</td>
<td>stfd</td>
<td>Store Floating-Point Double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>55 DC000000</td>
<td>139 FP</td>
<td>stfdu</td>
<td>Store Floating-Point Double with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQ</td>
<td>56 E0000000</td>
<td>58 LSQ</td>
<td>lq</td>
<td>Load Quadword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>57 E4000000</td>
<td>141 FP.out</td>
<td>lfdp</td>
<td>Load Floating-Point Double Pair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>58 E8000000</td>
<td>53 64</td>
<td>ld</td>
<td>Load Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>58 E8000001</td>
<td>53 64</td>
<td>ldu</td>
<td>Load Doubleword with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>58 E8000002</td>
<td>52 64</td>
<td>lwa</td>
<td>Load Word Algebraic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000044</td>
<td>289 DFP</td>
<td>dadd[,]</td>
<td>Decimal Floating Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC000064</td>
<td>300 DFP</td>
<td>dqual[,]</td>
<td>Decimal Quantize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC000024</td>
<td>145 FP</td>
<td>R</td>
<td>fdivs[,]</td>
<td>Floating Divide Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC000028</td>
<td>144 FP</td>
<td>R</td>
<td>fsubs[,]</td>
<td>Floating Subtract Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC00002A</td>
<td>144 FP</td>
<td>R</td>
<td>fadds[,]</td>
<td>Floating Add Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC00002C</td>
<td>146 FP</td>
<td>R</td>
<td>fsgrts[,]</td>
<td>Floating Square Root Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC000030</td>
<td>146 FP</td>
<td>R</td>
<td>fres[,]</td>
<td>Floating Reciprocal Estimate Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC000032</td>
<td>145 FP</td>
<td>R</td>
<td>fmuls[,]</td>
<td>Floating Multiply Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC000034</td>
<td>147 FP</td>
<td>R,ln</td>
<td>frsqrtes[,]</td>
<td>Floating Reciprocal Square Root Estimate Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC000038</td>
<td>149 FP</td>
<td>R</td>
<td>fsmuls[,]</td>
<td>Floating Multiply-Add Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC00003A</td>
<td>149 FP</td>
<td>R</td>
<td>fma[,]</td>
<td>Floating Add Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC00003C</td>
<td>150 FP</td>
<td>R</td>
<td>fnma[,]</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>59 EC00003E</td>
<td>150 FP</td>
<td>R</td>
<td>fnmadds[,]</td>
<td>Floating Negative Multiply-Add Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000044</td>
<td>291 DFP</td>
<td>dmuls[,]</td>
<td>Decimal Floating Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC000046</td>
<td>302 DFP</td>
<td>dmrd[,]</td>
<td>Decimal Floating Reround</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC000084</td>
<td>316 DFP</td>
<td>dscl[,]</td>
<td>Decimal Floating Shift Coefficient Left Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC000086</td>
<td>299 DFP</td>
<td>dqual[,]</td>
<td>Decimal Quantize Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC0000C4</td>
<td>316 DFP</td>
<td>dscl[,]</td>
<td>Decimal Floating Shift Coefficient Right Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC0000C6</td>
<td>305 DFP</td>
<td>drint[,]</td>
<td>Decimal Floating Round To FP Integer With Inexact</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000104</td>
<td>295 DFP</td>
<td>dcmpo</td>
<td>Decimal Floating Compare Ordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000144</td>
<td>297 DFP</td>
<td>dtstex</td>
<td>Decimal Floating Test Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC000184</td>
<td>296 DFP</td>
<td>dtstdc</td>
<td>Decimal Floating Test Data Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC0001C4</td>
<td>296 DFP</td>
<td>dtstdg</td>
<td>Decimal Floating Test Data Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC0001C6</td>
<td>307 DFP</td>
<td>drint[,]</td>
<td>Decimal Floating Round To FP Integer Without Inexact</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000204</td>
<td>309 DFP</td>
<td>dctdp[,]</td>
<td>Decimal Floating Convert To DFP Long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000244</td>
<td>311 DFP</td>
<td>dctfix[,]</td>
<td>Decimal Floating Convert To Fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000284</td>
<td>313 DFP</td>
<td>ddedpd[,]</td>
<td>Decimal Floating Decode DPD To BCD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC0002C4</td>
<td>314 DFP</td>
<td>dlex[,]</td>
<td>Decimal Floating Extract Exponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000404</td>
<td>289 DFP</td>
<td>dsab[,]</td>
<td>Decimal Floating Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000444</td>
<td>292 DFP</td>
<td>ddiv[,]</td>
<td>Decimal Floating Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000504</td>
<td>294 DFP</td>
<td>dcmpu</td>
<td>Decimal Floating Compare Unordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000544</td>
<td>298 DFP</td>
<td>dtstf</td>
<td>Decimal Floating Test Significance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000604</td>
<td>310 DFP</td>
<td>drsp[,]</td>
<td>Decimal Floating Round To DFP Short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000644</td>
<td>311 DFP</td>
<td>dcfix[,]</td>
<td>Decimal Floating Convert From Fixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000684</td>
<td>313 DFP</td>
<td>denbcd[,]</td>
<td>Decimal Floating Encode BCD To DPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC00069C</td>
<td>156 FP</td>
<td>R</td>
<td>fcfids[,]</td>
<td>Floating Convert From Integer Doubleword Single</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Primary</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC0006C4</td>
<td>DFP</td>
<td></td>
<td>314</td>
<td>die{[.]}</td>
</tr>
<tr>
<td>X</td>
<td>59</td>
<td>EC00079C</td>
<td>FP[R]</td>
<td></td>
<td>157</td>
<td>fcfidu{[.]}</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000000</td>
<td>VSX</td>
<td></td>
<td>405</td>
<td>xsaddsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000008</td>
<td>VSX</td>
<td></td>
<td>432</td>
<td>xsaddasp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000010</td>
<td>VSX</td>
<td></td>
<td>586</td>
<td>xssldwi</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000028</td>
<td>VSX</td>
<td></td>
<td>472</td>
<td>xssqrtesp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F000002C</td>
<td>VSX</td>
<td></td>
<td>474</td>
<td>xssqrtsp</td>
</tr>
<tr>
<td>XX4</td>
<td>60</td>
<td>F0000030</td>
<td>VSX</td>
<td></td>
<td>585</td>
<td>xsel</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000040</td>
<td>VSX</td>
<td></td>
<td>477</td>
<td>xssubsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000048</td>
<td>VSX</td>
<td></td>
<td>432</td>
<td>xsmaddssp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000050</td>
<td>VSX</td>
<td></td>
<td>585</td>
<td>xxpermddi</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000068</td>
<td>VSX</td>
<td></td>
<td>469</td>
<td>xsresp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000080</td>
<td>VSX</td>
<td></td>
<td>447</td>
<td>xsmulsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000088</td>
<td>VSX</td>
<td></td>
<td>442</td>
<td>xsmsubasp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000090</td>
<td>VSX</td>
<td></td>
<td>584</td>
<td>xxmrghw</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00000C0</td>
<td>VSX</td>
<td></td>
<td>427</td>
<td>xsdivsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00000C8</td>
<td>VSX</td>
<td></td>
<td>442</td>
<td>xsmsubmsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000100</td>
<td>VSX</td>
<td></td>
<td>400</td>
<td>xsadddp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000108</td>
<td>VSX</td>
<td></td>
<td>429</td>
<td>xsmaddadp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000118</td>
<td>VSX</td>
<td></td>
<td>409</td>
<td>xscmpudp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000120</td>
<td>VSX</td>
<td></td>
<td>418</td>
<td>xscvdpuxws</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000124</td>
<td>VSX</td>
<td></td>
<td>464</td>
<td>xsrdpi</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000128</td>
<td>VSX</td>
<td></td>
<td>471</td>
<td>xsrsqrtedp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F000012C</td>
<td>VSX</td>
<td></td>
<td>473</td>
<td>xssqrtdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000140</td>
<td>VSX</td>
<td></td>
<td>475</td>
<td>xssubdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000148</td>
<td>VSX</td>
<td></td>
<td>429</td>
<td>xsmaddmdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000158</td>
<td>VSX</td>
<td></td>
<td>407</td>
<td>xscmpodp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000160</td>
<td>VSX</td>
<td></td>
<td>413</td>
<td>xscvdpuxws</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000164</td>
<td>VSX</td>
<td></td>
<td>467</td>
<td>xsrdpiz</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000168</td>
<td>VSX</td>
<td></td>
<td>468</td>
<td>xsredp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000180</td>
<td>VSX</td>
<td></td>
<td>445</td>
<td>xsmuldp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000188</td>
<td>VSX</td>
<td></td>
<td>439</td>
<td>xsmsubadp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000190</td>
<td>VSX</td>
<td></td>
<td>584</td>
<td>xxmrghw</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00001A4</td>
<td>VSX</td>
<td></td>
<td>466</td>
<td>xsrdpip</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00001A8</td>
<td>VSX</td>
<td></td>
<td>480</td>
<td>xstsqrtdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00001AC</td>
<td>VSX</td>
<td></td>
<td>465</td>
<td>xsrdpic</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>XX3</td>
<td>F00001C0</td>
<td>425</td>
<td>VSX</td>
<td>60</td>
<td>xxdivdp</td>
<td>VSX Scalar Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F00001C8</td>
<td>439</td>
<td>VSX</td>
<td>60</td>
<td>xsmsubmdp</td>
<td>VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F00001E4</td>
<td>466</td>
<td>VSX</td>
<td>60</td>
<td>xsrdpim</td>
<td>VSX Scalar Round to Double-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>XX3</td>
<td>F00001E8</td>
<td>479</td>
<td>VSX</td>
<td>60</td>
<td>xstdivdp</td>
<td>VSX Scalar Test for software Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000200</td>
<td>486</td>
<td>VSX</td>
<td>60</td>
<td>xvaddsp</td>
<td>VSX Vector Add Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000208</td>
<td>521</td>
<td>VSX</td>
<td>60</td>
<td>xvmaddasp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000218</td>
<td>489</td>
<td>VSX</td>
<td>60</td>
<td>xcmeqsp</td>
<td>VSX Vector Compare Equal To Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F0000220</td>
<td>511</td>
<td>VSX</td>
<td>60</td>
<td>xvcvpsuxws</td>
<td>VSX Vector Convert Single-Precision to Unsigned Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>F0000224</td>
<td>566</td>
<td>VSX</td>
<td>60</td>
<td>xvrspi</td>
<td>VSX Vector Round to Single-Precision Integer</td>
</tr>
<tr>
<td>XX2</td>
<td>F0000228</td>
<td>570</td>
<td>VSX</td>
<td>60</td>
<td>xvsqrtsp</td>
<td>VSX Vector Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F000022C</td>
<td>572</td>
<td>VSX</td>
<td>60</td>
<td>xvsqrtsp</td>
<td>VSX Vector Square Root Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000240</td>
<td>575</td>
<td>VSX</td>
<td>60</td>
<td>xvsbsp</td>
<td>VSX Vector Subtract Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000248</td>
<td>524</td>
<td>VSX</td>
<td>60</td>
<td>xvmaddmsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000258</td>
<td>493</td>
<td>VSX</td>
<td>60</td>
<td>xvcmpgtsp</td>
<td>VSX Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F0000260</td>
<td>507</td>
<td>VSX</td>
<td>60</td>
<td>xvcvspxws</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>F0000264</td>
<td>568</td>
<td>VSX</td>
<td>60</td>
<td>xvrspiz</td>
<td>VSX Vector Round to Single-Precision Integer toward Zero</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000268</td>
<td>565</td>
<td>VSX</td>
<td>60</td>
<td>xvresp</td>
<td>VSX Vector Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000280</td>
<td>543</td>
<td>VSX</td>
<td>60</td>
<td>xvmslp</td>
<td>VSX Vector Multiply Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000288</td>
<td>535</td>
<td>VSX</td>
<td>60</td>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000290</td>
<td>586</td>
<td>VSX</td>
<td>60</td>
<td>xxspltw</td>
<td>VSX Splat Word</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000298</td>
<td>491</td>
<td>VSX</td>
<td>60</td>
<td>xvcmpgesp</td>
<td>VSX Vector Compare Greater Than or Equal To Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F00002A0</td>
<td>516</td>
<td>VSX</td>
<td>60</td>
<td>xvcvuxwsp</td>
<td>VSX Vector Convert Unsigned Fixed-Point Word to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F00002A4</td>
<td>567</td>
<td>VSX</td>
<td>60</td>
<td>xvrspip</td>
<td>VSX Vector Round to Single-Precision Integer toward +Infinity</td>
</tr>
<tr>
<td>XX2</td>
<td>F00002A8</td>
<td>579</td>
<td>VSX</td>
<td>60</td>
<td>xvtsqrtsp</td>
<td>VSX Vector Test for software Square Root Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F00002AC</td>
<td>566</td>
<td>VSX</td>
<td>60</td>
<td>xvrspic</td>
<td>VSX Vector Round to Single-Precision Integer using Current rounding mode</td>
</tr>
<tr>
<td>XX3</td>
<td>F00002C0</td>
<td>519</td>
<td>VSX</td>
<td>60</td>
<td>xdivsp</td>
<td>VSX Vector Divide Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F00002C8</td>
<td>538</td>
<td>VSX</td>
<td>60</td>
<td>xvmsubmsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F00002E0</td>
<td>514</td>
<td>VSX</td>
<td>60</td>
<td>xvcsvxwsp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>F00002E4</td>
<td>567</td>
<td>VSX</td>
<td>60</td>
<td>xvrspim</td>
<td>VSX Vector Round to Single-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>XX3</td>
<td>F00002E8</td>
<td>578</td>
<td>VSX</td>
<td>60</td>
<td>xvtdivsp</td>
<td>VSX Vector Test for software Divide Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000300</td>
<td>482</td>
<td>VSX</td>
<td>60</td>
<td>xvadddp</td>
<td>VSX Vector Add Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000308</td>
<td>521</td>
<td>VSX</td>
<td>60</td>
<td>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>F0000318</td>
<td>488</td>
<td>VSX</td>
<td>60</td>
<td>xvcmeqdp</td>
<td>VSX Vector Compare Equal To Double-Precision</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000320</td>
<td>502</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000320</td>
<td>502</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000324</td>
<td>561</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000324</td>
<td>561</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000328</td>
<td>568</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000328</td>
<td>568</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000032C</td>
<td>571</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F000032C</td>
<td>571</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000340</td>
<td>573</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000340</td>
<td>573</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000348</td>
<td>524</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000348</td>
<td>524</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000358</td>
<td>492</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000358</td>
<td>492</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000360</td>
<td>498</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000360</td>
<td>498</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000364</td>
<td>563</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000364</td>
<td>563</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000368</td>
<td>564</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000368</td>
<td>564</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000380</td>
<td>541</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000380</td>
<td>541</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000388</td>
<td>535</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000388</td>
<td>535</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000398</td>
<td>490</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000398</td>
<td>490</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003A0</td>
<td>516</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00003A0</td>
<td>516</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003A4</td>
<td>562</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00003A4</td>
<td>562</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003A8</td>
<td>579</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00003A8</td>
<td>579</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003AC</td>
<td>561</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00003AC</td>
<td>561</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003C0</td>
<td>517</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F00003C0</td>
<td>517</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003C8</td>
<td>538</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F00003C8</td>
<td>538</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003E0</td>
<td>514</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00003E0</td>
<td>514</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003E4</td>
<td>562</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00003E4</td>
<td>562</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003E8</td>
<td>577</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F00003E8</td>
<td>577</td>
</tr>
<tr>
<td>VX</td>
<td>60 F0000400</td>
<td>267</td>
<td>V</td>
<td>VX</td>
<td>60 F0000400</td>
<td>267</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000408</td>
<td>455</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000408</td>
<td>455</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000410</td>
<td>580</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000410</td>
<td>580</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000424</td>
<td>412</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000424</td>
<td>412</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000042C</td>
<td>413</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F000042C</td>
<td>413</td>
</tr>
<tr>
<td>VX</td>
<td>60 F0000440</td>
<td>267</td>
<td>V</td>
<td>VX</td>
<td>60 F0000440</td>
<td>267</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000448</td>
<td>455</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000448</td>
<td>455</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000450</td>
<td>580</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000450</td>
<td>580</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000464</td>
<td>470</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F0000464</td>
<td>470</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000488</td>
<td>461</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000488</td>
<td>461</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000490</td>
<td>583</td>
<td>VSX</td>
<td>xx3</td>
<td>60 F0000490</td>
<td>583</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00004A0</td>
<td>424</td>
<td>VSX</td>
<td>xx2</td>
<td>60 F00004A0</td>
<td>424</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00004C8</td>
<td></td>
<td>461</td>
<td>VSX</td>
<td>xsnmsubmssp VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00004D0</td>
<td></td>
<td>583</td>
<td>VSX</td>
<td>xxixor VSX Logical XOR</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00004E0</td>
<td></td>
<td>423</td>
<td>VSX</td>
<td>xscvsxdsp VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000500</td>
<td></td>
<td>435</td>
<td>VSX</td>
<td>xsmmaxd VSX Scalar Maximum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000508</td>
<td></td>
<td>450</td>
<td>VSX</td>
<td>xsnmaddadp VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000510</td>
<td></td>
<td>582</td>
<td>VSX</td>
<td>xxlnor VSX Logical NOR</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000520</td>
<td></td>
<td>416</td>
<td>VSX</td>
<td>xscvdpuxds VSX Scalar Convert Double-Precision to Signed Fixed-Point Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000524</td>
<td></td>
<td>420</td>
<td>VSX</td>
<td>xscvsdp VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F000052C</td>
<td></td>
<td>422</td>
<td>VSX</td>
<td>xscvspdpp VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000540</td>
<td></td>
<td>437</td>
<td>VSX</td>
<td>xsminmp VSX Scalar Minimum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000548</td>
<td></td>
<td>450</td>
<td>VSX</td>
<td>xsnmaddmp VSX Scalar Negative Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000550</td>
<td></td>
<td>582</td>
<td>VSX</td>
<td>xxlorc VSX Logical OR with Complement</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000560</td>
<td></td>
<td>422</td>
<td>VSX</td>
<td>xscvdxpxds VSX Scalar Convert Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000564</td>
<td></td>
<td>399</td>
<td>VSX</td>
<td>xsabspd VSX Scalar Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000580</td>
<td></td>
<td>411</td>
<td>VSX</td>
<td>xscpsgdp VSX Scalar Copy Sign Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000588</td>
<td></td>
<td>458</td>
<td>VSX</td>
<td>xsnmsubadp VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000590</td>
<td></td>
<td>581</td>
<td>VSX</td>
<td>xxlnand VSX Logical NAND</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00005A0</td>
<td></td>
<td>424</td>
<td>VSX</td>
<td>xscvdxpdp VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00005A4</td>
<td></td>
<td>449</td>
<td>VSX</td>
<td>xsnabsdp VSX Scalar Negative Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00005C8</td>
<td></td>
<td>458</td>
<td>VSX</td>
<td>xsnmsubadp VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00005D0</td>
<td></td>
<td>581</td>
<td>VSX</td>
<td>xxleqv VSX Logical Equivalence</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00005E0</td>
<td></td>
<td>423</td>
<td>VSX</td>
<td>xscvsxdpp VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00005E4</td>
<td></td>
<td>449</td>
<td>VSX</td>
<td>xsnegdp VSX Scalar Negate Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000600</td>
<td></td>
<td>529</td>
<td>VSX</td>
<td>xvmaxsp VSX Vector Maximum Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000608</td>
<td></td>
<td>547</td>
<td>VSX</td>
<td>xvmaddasp VSX Vector Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000618</td>
<td></td>
<td>489</td>
<td>VSX</td>
<td>xvcmpqesp VSX Vector Compare Equal To Single-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000620</td>
<td></td>
<td>509</td>
<td>VSX</td>
<td>xvcvpsuxds VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000624</td>
<td></td>
<td>495</td>
<td>VSX</td>
<td>xvcvdpdp VSX Vector Convert Double-Precision to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000640</td>
<td></td>
<td>533</td>
<td>VSX</td>
<td>xvminsp VSX Vector Minimum Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000648</td>
<td></td>
<td>552</td>
<td>VSX</td>
<td>xvmaddmsp VSX Vector Negative Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000658</td>
<td></td>
<td>493</td>
<td>VSX</td>
<td>xvcmpagsp VSX Vector Compare Greater Than Single-Precision &amp; record CR6</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000660</td>
<td></td>
<td>505</td>
<td>VSX</td>
<td>xvcvpsuxds VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000664</td>
<td></td>
<td>481</td>
<td>VSX</td>
<td>xvbssp VSX Vector Absolute Value Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000680</td>
<td></td>
<td>494</td>
<td>VSX</td>
<td>xvcpsgsp VSX Vector Copy Sign Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000688</td>
<td></td>
<td>555</td>
<td>VSX</td>
<td>xvmmsubasp VSX Vector Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000698</td>
<td></td>
<td>491</td>
<td>VSX</td>
<td>xvcmpgesp VSX Vector Compare Greater Than or Equal To Single-Precision &amp; record CR6</td>
</tr>
<tr>
<td>Format</td>
<td>Primary Base (hex)</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006A0</td>
<td></td>
<td></td>
<td>515</td>
<td>VSX</td>
<td>xcvvuxdsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006A4</td>
<td></td>
<td></td>
<td>545</td>
<td>VSX</td>
<td>xvnabssp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00006C8</td>
<td></td>
<td></td>
<td>558</td>
<td>VSX</td>
<td>xvnmsubmsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006E0</td>
<td></td>
<td></td>
<td>513</td>
<td>VSX</td>
<td>xvcvxsdsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006E4</td>
<td></td>
<td></td>
<td>546</td>
<td>VSX</td>
<td>xvnegsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000700</td>
<td></td>
<td></td>
<td>527</td>
<td>VSX</td>
<td>xvmaxdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000708</td>
<td></td>
<td></td>
<td>547</td>
<td>VSX</td>
<td>xvmaddadp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000718</td>
<td></td>
<td></td>
<td>488</td>
<td>VSX</td>
<td>xvcmpqdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000720</td>
<td></td>
<td></td>
<td>500</td>
<td>VSX</td>
<td>xvcvdpuxds</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000724</td>
<td></td>
<td></td>
<td>504</td>
<td>VSX</td>
<td>xvcvspdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000740</td>
<td></td>
<td></td>
<td>531</td>
<td>VSX</td>
<td>xvmindp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000748</td>
<td></td>
<td></td>
<td>552</td>
<td>VSX</td>
<td>xvmaddmdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000758</td>
<td></td>
<td></td>
<td>492</td>
<td>VSX</td>
<td>xvcmpgtdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000760</td>
<td></td>
<td></td>
<td>496</td>
<td>VSX</td>
<td>xvcvdpuxds</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000764</td>
<td></td>
<td></td>
<td>480</td>
<td>VSX</td>
<td>xvabsdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000780</td>
<td></td>
<td></td>
<td>494</td>
<td>VSX</td>
<td>xvcpsgdnp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000788</td>
<td></td>
<td></td>
<td>555</td>
<td>VSX</td>
<td>xvnmsubadp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000798</td>
<td></td>
<td></td>
<td>490</td>
<td>VSX</td>
<td>xvcmpgedp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007A0</td>
<td></td>
<td></td>
<td>515</td>
<td>VSX</td>
<td>xcvvuxdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007A4</td>
<td></td>
<td></td>
<td>545</td>
<td>VSX</td>
<td>xvnabsdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00007C8</td>
<td></td>
<td></td>
<td>558</td>
<td>VSX</td>
<td>xvnmsubmdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007E0</td>
<td></td>
<td></td>
<td>513</td>
<td>VSX</td>
<td>xvcvxsddp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007E4</td>
<td></td>
<td></td>
<td>546</td>
<td>VSX</td>
<td>xvnegdp</td>
</tr>
<tr>
<td>DS</td>
<td>61 F4000000</td>
<td></td>
<td></td>
<td>141</td>
<td>FP</td>
<td>stfdp</td>
</tr>
<tr>
<td>DS</td>
<td>62 F8000000</td>
<td>57</td>
<td>64</td>
<td></td>
<td></td>
<td>std</td>
</tr>
<tr>
<td>DS</td>
<td>62 F8000001</td>
<td>57</td>
<td>64</td>
<td></td>
<td></td>
<td>stdu</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00000EE</td>
<td></td>
<td></td>
<td>159</td>
<td>FP</td>
<td>fcmpu</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00000F</td>
<td></td>
<td></td>
<td>289</td>
<td>DFP</td>
<td>daddq[,]</td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC000006</td>
<td></td>
<td></td>
<td>300</td>
<td>DFP</td>
<td>dquaq[,]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000010</td>
<td></td>
<td></td>
<td>142</td>
<td>FP[R]</td>
<td>fcpsgn[,]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000018</td>
<td></td>
<td></td>
<td>151</td>
<td>FP[R]</td>
<td>frsl[,]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00001C</td>
<td></td>
<td></td>
<td>153</td>
<td>FP[R]</td>
<td>fcntlw[,]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00001E</td>
<td></td>
<td></td>
<td>154</td>
<td>FP[R]</td>
<td>fctiwz[,]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC000024</td>
<td></td>
<td></td>
<td>145</td>
<td>FP[R]</td>
<td>fdiv[,]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC000028</td>
<td></td>
<td></td>
<td>144</td>
<td>FP[R]</td>
<td>fsub[,]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC00002A</td>
<td></td>
<td></td>
<td>144</td>
<td>FP[R]</td>
<td>fadd[,]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC00002C</td>
<td></td>
<td></td>
<td>146</td>
<td>FP[R]</td>
<td>fsqrt[,]</td>
</tr>
<tr>
<td>A</td>
<td>63 FC00002E</td>
<td></td>
<td></td>
<td>160</td>
<td>FP[R]</td>
<td>fsel[,]</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>A</td>
<td>63 FC000030</td>
<td>146</td>
<td>FP[R].in</td>
<td>fre[]</td>
<td>Floating Reciprocal Estimate</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>63 FC000032</td>
<td>145</td>
<td>FP[R]</td>
<td>fmul[]</td>
<td>Floating Multiply</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>63 FC000034</td>
<td>147</td>
<td>FP[R]</td>
<td>fsqrt[]</td>
<td>Floating Reciprocal Square Root Estimate</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>63 FC000038</td>
<td>149</td>
<td>FP[R]</td>
<td>fsub[]</td>
<td>Floating Multiply-Subtract</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>63 FC00003A</td>
<td>149</td>
<td>FP[R]</td>
<td>fmadd[]</td>
<td>Floating Multiply-Add</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>63 FC00003C</td>
<td>150</td>
<td>FP[R]</td>
<td>fnmsub[]</td>
<td>Floating Negative Multiply-Subtract</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>63 FC00003E</td>
<td>150</td>
<td>FP[R]</td>
<td>fnmadd[]</td>
<td>Floating Negative Multiply-Add</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000040</td>
<td>159</td>
<td>FP</td>
<td>fcmp[]</td>
<td>Floating Compare Ordered</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000044</td>
<td>291</td>
<td>DFP</td>
<td>dmul[]</td>
<td>Decimal Floating Multiply Quad</td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC000046</td>
<td>302</td>
<td>DFP</td>
<td>drmd[]</td>
<td>Decimal Floating Round Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00004C</td>
<td>163</td>
<td>FP[R]</td>
<td>mtfsb[]</td>
<td>Move To FPSCR Bit 1</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000050</td>
<td>142</td>
<td>FP[R]</td>
<td>fneg[]</td>
<td>Floating Negate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000080</td>
<td>161</td>
<td>FP</td>
<td>mcfs[]</td>
<td>Move To Condition Register from FPSCR</td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC000084</td>
<td>316</td>
<td>DFP</td>
<td>dscl[]</td>
<td>Decimal Floating Shift Coefficient Left Immediate Quad</td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC000086</td>
<td>299</td>
<td>DFP</td>
<td>dqua[]</td>
<td>Decimal Quantize Immediate Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00008C</td>
<td>163</td>
<td>FP[R]</td>
<td>mtfsb[]</td>
<td>Move To FPSCR Bit 0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000090</td>
<td>142</td>
<td>FP[R]</td>
<td>fnsr[]</td>
<td>Floating Move Register</td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC0000C4</td>
<td>316</td>
<td>DFP</td>
<td>dscri[]</td>
<td>Decimal Floating Shift Coefficient Right Immediate Quad</td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC0000C6</td>
<td>305</td>
<td>DFP</td>
<td>drint[]</td>
<td>Decimal Floating Round To FP Integer Without Inexact Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000100</td>
<td>148</td>
<td>FP</td>
<td>fdiv[]</td>
<td>Floating Test for software Divide</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000104</td>
<td>295</td>
<td>DFP</td>
<td>dcmp[]</td>
<td>Decimal Floating Compare Ordered Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00010C</td>
<td>162</td>
<td>FP[R]</td>
<td>mtsf[]</td>
<td>Move To FPSCR Field Immediate</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000110</td>
<td>142</td>
<td>FP[R]</td>
<td>fnabs[]</td>
<td>Floating Negative Absolute Value</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00011C</td>
<td>154</td>
<td>FP[R]</td>
<td>fctiw[]</td>
<td>Floating Convert To Integer Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00011E</td>
<td>155</td>
<td>FP[R]</td>
<td>fctiwu[]</td>
<td>Floating Convert To Integer Word Unsigned with round toward Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000140</td>
<td>148</td>
<td>FP</td>
<td>ftqrt[]</td>
<td>Floating Test for software Square Root</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000144</td>
<td>297</td>
<td>DFP</td>
<td>dstex[]</td>
<td>Decimal Floating Test Exponent Quad</td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC000184</td>
<td>296</td>
<td>DFP</td>
<td>dstsd[]</td>
<td>Decimal Floating Test Data Class Quad</td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC0001C4</td>
<td>296</td>
<td>DFP</td>
<td>dstsd[]</td>
<td>Decimal Floating Test Data Group Quad</td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC0001C6</td>
<td>307</td>
<td>DFP</td>
<td>drint[]</td>
<td>Decimal Floating Round To FP Integer Without Inexact Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000204</td>
<td>309</td>
<td>DFP</td>
<td>dctpq[]</td>
<td>Decimal Floating Convert To DFP Extended</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000210</td>
<td>142</td>
<td>FP[R]</td>
<td>fab[]</td>
<td>Floating Absolute Value</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000244</td>
<td>311</td>
<td>DFP</td>
<td>dfctx[]</td>
<td>Decimal Floating Convert To Fixed Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000284</td>
<td>313</td>
<td>DFP</td>
<td>ddendp[]</td>
<td>Decimal Floating Decode DPD To BCD Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC0002C4</td>
<td>314</td>
<td>DFP</td>
<td>dxex[]</td>
<td>Decimal Floating Extract Exponent Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000310</td>
<td>158</td>
<td>FP[R].in</td>
<td>fmr[]</td>
<td>Floating Round To Integer Nearest</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000350</td>
<td>158</td>
<td>FP[R].in</td>
<td>frit[]</td>
<td>Floating Round To Integer toward Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000390</td>
<td>158</td>
<td>FP[R].in</td>
<td>frt[]</td>
<td>Floating Round To Integer Plus</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC0003D0</td>
<td>158</td>
<td>FP[R].in</td>
<td>frm[]</td>
<td>Floating Round To Integer Minus</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000404</td>
<td>289</td>
<td>DFP</td>
<td>dsu[]</td>
<td>Decimal Floating Subtract Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000444</td>
<td>292</td>
<td>DFP</td>
<td>ddiv[]</td>
<td>Decimal Floating Divide Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00048E</td>
<td>161</td>
<td>FP[R]</td>
<td>mfs[]</td>
<td>Move From FPSCR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000504</td>
<td>295</td>
<td>DFP</td>
<td>dcmpu[]</td>
<td>Decimal Floating Compare Unordered Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000544</td>
<td>298</td>
<td>DFP</td>
<td>dstst[]</td>
<td>Decimal Floating Test Significance Quad</td>
<td></td>
</tr>
<tr>
<td>XFL</td>
<td>63 FC00058E</td>
<td>162</td>
<td>FP[R]</td>
<td>mtsf[]</td>
<td>Move To FPSCR Fields</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000604</td>
<td>310</td>
<td>DFP</td>
<td>drdp[]</td>
<td>Decimal Floating Round To DFP Long</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000644</td>
<td>311</td>
<td>DFP</td>
<td>dfctx[]</td>
<td>Decimal Floating Convert From Fixed Quad</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00065C</td>
<td>151</td>
<td>FP[R]</td>
<td>fcbd[]</td>
<td>Floating Convert To Integer Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00065E</td>
<td>152</td>
<td>FP[R]</td>
<td>fctidz[]</td>
<td>Floating Convert To Integer Doubleword with round toward Zero</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Primary Base (hex)</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000684</td>
<td>313</td>
<td>DFP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00068C</td>
<td>143</td>
<td>VSX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00069C</td>
<td>155</td>
<td>FP[R]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC0006C4</td>
<td>314</td>
<td>DFP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00075C</td>
<td>152</td>
<td>FP[R]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00075E</td>
<td>153</td>
<td>FP[R]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00078C</td>
<td>143</td>
<td>VSX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00079C</td>
<td>156</td>
<td>FP[R]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.
# Appendix I. Power ISA Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the Power ISA, sorted by mnemonic.

<table>
<thead>
<tr>
<th>Format</th>
<th>Primary Base (hex)</th>
<th>Mode Dep.¹</th>
<th>Privilege¹</th>
<th>Page</th>
<th>Category¹</th>
<th>Mnemonic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>XO</td>
<td>31 7C00214</td>
<td>SR</td>
<td>B</td>
<td>67</td>
<td>add[,]</td>
<td>Add</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00014</td>
<td>SR</td>
<td>B</td>
<td>68</td>
<td>addc[,]</td>
<td>AddCarrying</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00414</td>
<td>SR</td>
<td>B</td>
<td>68</td>
<td>addco[,]</td>
<td>AddCarrying &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00014</td>
<td>SR</td>
<td>B</td>
<td>69</td>
<td>ade[,]</td>
<td>Add Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000514</td>
<td>SR</td>
<td>B</td>
<td>69</td>
<td>adeo[,]</td>
<td>Add Extended &amp; record OV &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000094</td>
<td>H</td>
<td>101</td>
<td>BCD</td>
<td>addg6s</td>
<td>Add and Generate Sixes</td>
</tr>
<tr>
<td>D</td>
<td>14 3B000000</td>
<td>B</td>
<td>B</td>
<td>66</td>
<td>addi</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>D</td>
<td>12 3B000000</td>
<td>SR</td>
<td>B</td>
<td>67</td>
<td>addic</td>
<td>Add Immediate Carrying</td>
</tr>
<tr>
<td>D</td>
<td>13 3B000000</td>
<td>SR</td>
<td>B</td>
<td>67</td>
<td>adic.</td>
<td>Add Immediate Carrying &amp; record CR0</td>
</tr>
<tr>
<td>D</td>
<td>15 3D000000</td>
<td>B</td>
<td>B</td>
<td>66</td>
<td>addis</td>
<td>Add Immediate Shifted</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001E4</td>
<td>SR</td>
<td>B</td>
<td>69</td>
<td>adme[,]</td>
<td>Add to Minus One Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0005D4</td>
<td>SR</td>
<td>B</td>
<td>69</td>
<td>admeo[,]</td>
<td>Add to Minus One Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000614</td>
<td>SR</td>
<td>B</td>
<td>67</td>
<td>adol[,]</td>
<td>Add &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000194</td>
<td>SR</td>
<td>B</td>
<td>70</td>
<td>adze[,]</td>
<td>Add to Zero Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000594</td>
<td>SR</td>
<td>B</td>
<td>70</td>
<td>adzeo[,]</td>
<td>Add to Zero Extended &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00038</td>
<td>SR</td>
<td>B</td>
<td>84</td>
<td>and[,]</td>
<td>AND</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00078</td>
<td>SR</td>
<td>B</td>
<td>85</td>
<td>andc[,]</td>
<td>AND with Complement</td>
</tr>
<tr>
<td>D</td>
<td>28 7C000000</td>
<td>SR</td>
<td>B</td>
<td>82</td>
<td>andi.</td>
<td>AND Immediate &amp; record CR0</td>
</tr>
<tr>
<td>D</td>
<td>29 7C000000</td>
<td>SR</td>
<td>B</td>
<td>82</td>
<td>andis.</td>
<td>AND Immediate Shifted &amp; record CR0</td>
</tr>
<tr>
<td>I</td>
<td>18 4C000000</td>
<td>B</td>
<td>B</td>
<td>38</td>
<td>bl[[a]</td>
<td>Branch</td>
</tr>
<tr>
<td>B</td>
<td>16 4C000000</td>
<td>CT</td>
<td>B</td>
<td>38</td>
<td>bc[][a]</td>
<td>Branch Conditional</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000420</td>
<td>CT</td>
<td>B</td>
<td>39</td>
<td>bcctr[l]</td>
<td>Branch Conditional to Count Register</td>
</tr>
<tr>
<td>VX</td>
<td>60 F0000400</td>
<td>V</td>
<td>B</td>
<td>267</td>
<td>bcaddd.</td>
<td>Decimal Add Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>60 F0000440</td>
<td>V</td>
<td>B</td>
<td>267</td>
<td>bcadsub.</td>
<td>Decimal Subtract Modulo</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000020</td>
<td>CT</td>
<td>B</td>
<td>39</td>
<td>bcrl[l]</td>
<td>Branch Conditional to Link Register</td>
</tr>
<tr>
<td>X</td>
<td>19 4C000460</td>
<td>B</td>
<td>B</td>
<td>40</td>
<td>bctar[l]</td>
<td>Branch Conditional to Branch Target Address Register</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001F8</td>
<td>B</td>
<td>B</td>
<td>90</td>
<td>bpermd</td>
<td>Bit Permute Doubleword</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000020F</td>
<td>SP</td>
<td>B</td>
<td>594</td>
<td>brinc</td>
<td>Bit Reversed Increment</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000274</td>
<td>H</td>
<td>101</td>
<td>BCD</td>
<td>cbcdtd</td>
<td>Convert Binary Coded Decimal To Declets</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000234</td>
<td>H</td>
<td>101</td>
<td>BCD</td>
<td>cdtdbcd</td>
<td>Convert Declets To Binary Coded Decimal</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00035C</td>
<td>S</td>
<td>B</td>
<td>44</td>
<td>clrhrb</td>
<td>Clear BHRB</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000000</td>
<td>B</td>
<td>B</td>
<td>78</td>
<td>cmp</td>
<td>Compare</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003F8</td>
<td>B</td>
<td>B</td>
<td>86</td>
<td>cmpb</td>
<td>Compare Byte</td>
</tr>
<tr>
<td>D</td>
<td>11 2C000000</td>
<td>B</td>
<td>B</td>
<td>78</td>
<td>cmpi</td>
<td>Compare Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000040</td>
<td>B</td>
<td>B</td>
<td>79</td>
<td>cmpl</td>
<td>Compare Logical</td>
</tr>
<tr>
<td>D</td>
<td>10 2B000000</td>
<td>B</td>
<td>B</td>
<td>79</td>
<td>cmpli</td>
<td>Compare Logical Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000074</td>
<td>SR</td>
<td>B</td>
<td>89</td>
<td>cntizd[,]</td>
<td>Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000334</td>
<td>SR</td>
<td>B</td>
<td>85</td>
<td>cntizw[,]</td>
<td>Count Leading Zeros Word</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000202</td>
<td>B</td>
<td>B</td>
<td>41</td>
<td>crand</td>
<td>Condition Register AND</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode Base (hex)</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000102</td>
<td>42 B</td>
<td>crandc</td>
<td></td>
<td></td>
<td>Condition Register AND with Complement</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000242</td>
<td>42 B</td>
<td>creqv</td>
<td></td>
<td></td>
<td>Condition Register Equivalent</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C0001C2</td>
<td>41 B</td>
<td>crnand</td>
<td></td>
<td></td>
<td>Condition Register NAND</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000042</td>
<td>42 B</td>
<td>crnor</td>
<td></td>
<td></td>
<td>Condition Register NOR</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C0003B2</td>
<td>41 B</td>
<td>cror</td>
<td></td>
<td></td>
<td>Condition Register OR</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000342</td>
<td>42 B</td>
<td>crorc</td>
<td></td>
<td></td>
<td>Condition Register OR with Complement</td>
</tr>
<tr>
<td>X</td>
<td>59 EC00004</td>
<td>289 DFP</td>
<td>dadd[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Add</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00004</td>
<td>289 DFP</td>
<td>daddq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Add Quad</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0005EC</td>
<td>770 E</td>
<td>dcba</td>
<td></td>
<td></td>
<td>Data Cache Block Allocate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00004C</td>
<td>773 B</td>
<td>dcbf</td>
<td></td>
<td></td>
<td>Data Cache Block Flush</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000FE</td>
<td>P 1064 E.PD</td>
<td>dcbfep</td>
<td></td>
<td></td>
<td>Data Cache Block Flush by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003AC</td>
<td>P 1118 E</td>
<td>dcbi</td>
<td></td>
<td></td>
<td>Data Cache Block Invalidate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00030C</td>
<td>M 1123 ECL</td>
<td>dcblc</td>
<td></td>
<td></td>
<td>Data Cache Block Lock Clear</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00034D</td>
<td>M 1121 ECL</td>
<td>dcblq.</td>
<td></td>
<td></td>
<td>Data Cache Block Lock Query</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00006C</td>
<td>773 B</td>
<td>dcbst</td>
<td></td>
<td></td>
<td>Data Cache Block Store</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00007E</td>
<td>P 1063 E.PD</td>
<td>dcbstep</td>
<td></td>
<td></td>
<td>Data Cache Block Store by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00022C</td>
<td>770 B</td>
<td>dcct</td>
<td></td>
<td></td>
<td>Data Cache Block Touch</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00027E</td>
<td>P 1063 E.PD</td>
<td>dcbstep</td>
<td></td>
<td></td>
<td>Data Cache Block Touch by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00014C</td>
<td>M 1122 ECL</td>
<td>dcblts</td>
<td></td>
<td></td>
<td>Data Cache Block Touch and Lock Set</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001EC</td>
<td>771 B</td>
<td>dcbtst</td>
<td></td>
<td></td>
<td>Data Cache Block Touch for Store</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001FE</td>
<td>P 1066 E.PD</td>
<td>dcbtstep</td>
<td></td>
<td></td>
<td>Data Cache Block Touch for Store by External PID</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00010C</td>
<td>M 1122 ECL</td>
<td>dcbtstls</td>
<td></td>
<td></td>
<td>Data Cache Block Touch for Store and Lock Set</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007EC</td>
<td>773 B</td>
<td>dcbtz</td>
<td></td>
<td></td>
<td>Data Cache Block Zero</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007FE</td>
<td>P 1067 E.PD</td>
<td>dcbzep</td>
<td></td>
<td></td>
<td>Data Cache Block Zero by External PID</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000644</td>
<td>311 DFP</td>
<td>dcfix[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Convert From Fixed</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000644</td>
<td>311 DFP</td>
<td>dcfixq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Convert From Fixed Quad</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003B8</td>
<td>P 1239 E.CI</td>
<td>dci</td>
<td></td>
<td></td>
<td>Data Cache Invalidate</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000104</td>
<td>295 DFP</td>
<td>dcmpo</td>
<td></td>
<td></td>
<td>Decimal Floating Compare Ordered</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000104</td>
<td>295 DFP</td>
<td>dcmpoq</td>
<td></td>
<td></td>
<td>Decimal Floating Compare Ordered Quad</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000504</td>
<td>294 DFP</td>
<td>dcmpo</td>
<td></td>
<td></td>
<td>Decimal Floating Compare Unordered</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000504</td>
<td>295 DFP</td>
<td>dcmpoq</td>
<td></td>
<td></td>
<td>Decimal Floating Compare Unordered Quad</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00028C</td>
<td>P 1242 E.CD</td>
<td>dcread</td>
<td></td>
<td></td>
<td>Data Cache Read</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003CC</td>
<td>P 1242 E.CD</td>
<td>dcread</td>
<td></td>
<td></td>
<td>Data Cache Read</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000204</td>
<td>309 DFP</td>
<td>dctdp[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Convert To DFP Long</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000244</td>
<td>311 DFP</td>
<td>dcflix[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Convert To Fixed</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000244</td>
<td>311 DFP</td>
<td>dcflixq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Convert To Fixed Quad</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000284</td>
<td>309 DFP</td>
<td>dctpq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Convert To DFP Extended</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000284</td>
<td>313 DFP</td>
<td>ddedpd[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Decode DPD To BCD</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000284</td>
<td>313 DFP</td>
<td>ddedpdq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Decode DPD To BCD Quad</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000444</td>
<td>292 DFP</td>
<td>ddv[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Divide</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000444</td>
<td>292 DFP</td>
<td>ddvq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Divide Quad</td>
</tr>
<tr>
<td>X</td>
<td>59 EC000684</td>
<td>313 DFP</td>
<td>dencbc[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Encode BCD To BCD</td>
</tr>
<tr>
<td>X</td>
<td>63 FC000684</td>
<td>313 DFP</td>
<td>dencbcq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Encode BCD To BCD Quad</td>
</tr>
<tr>
<td>X</td>
<td>59 EC0006C4</td>
<td>314 DFP</td>
<td>die[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Insert Exponent</td>
</tr>
<tr>
<td>X</td>
<td>63 FC0006C4</td>
<td>314 DFP</td>
<td>dieq[.]</td>
<td></td>
<td></td>
<td>Decimal Floating Insert Exponent Quad</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0003D2</td>
<td>SR 76 64</td>
<td>ddiv[.]</td>
<td></td>
<td></td>
<td>Divide Doubleword</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000352</td>
<td>SR 77 64</td>
<td>ddiv[.]</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000752</td>
<td>SR 77 64</td>
<td>ddivdeo[.]</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000312</td>
<td>SR 77 64</td>
<td>ddivreu[.]</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended Unsigned</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000712</td>
<td>SR 77 64</td>
<td>ddivdeo[.]</td>
<td></td>
<td></td>
<td>Divide Doubleword Extended Unsigned &amp; record OV</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Page</td>
<td>Mnemonic Instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------</td>
<td>----------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0007D2 SR 76</td>
<td>64</td>
<td>divdow[] Divide Doubleword &amp; record OV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000392 SR 76</td>
<td>64</td>
<td>divdu[] Divide Doubleword Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000792 SR 76</td>
<td>64</td>
<td>divduo[] Divide Doubleword Unsigned &amp; record OV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0003D6 SR 72</td>
<td>B</td>
<td>divw[] Divide Word</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000356 SR 73</td>
<td>B</td>
<td>divwe[] Divide Word Extended</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000756 SR 73</td>
<td>B</td>
<td>divweo[] Divide Word Extended &amp; record OV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000316 SR 72</td>
<td>B</td>
<td>divweu[] Divide Word Extended Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000716 SR 72</td>
<td>B</td>
<td>divweuo[] Divide Word Extended Unsigned &amp; record OV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000396 SR 72</td>
<td>B</td>
<td>divwu[] Divide Word Unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000796 SR 72</td>
<td>B</td>
<td>divwu[] Divide Word Unsigned &amp; record OV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00009C</td>
<td>673</td>
<td>LMV dlmzb[] Determine Leftmost Zero Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000044</td>
<td>291</td>
<td>DFP dmul[] Decimal Floating Multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000044</td>
<td>291</td>
<td>DFP dmulq[] Decimal Floating Multiply Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC00018C</td>
<td>1228</td>
<td>E dnh Debugger Notify Halt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000324</td>
<td>H 867</td>
<td>S doze Doze</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC000006</td>
<td>300</td>
<td>DFP dqua[] Decimal Quantize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC000086</td>
<td>299</td>
<td>DFP dqua[] Decimal Quantize Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC000006</td>
<td>300</td>
<td>DFP dquaq[] Decimal Quantize Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000064</td>
<td>310</td>
<td>DFP drdpq[] Decimal Floating Round To DFP Long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC0001C6</td>
<td>307</td>
<td>DFP drintn[] Decimal Floating Round To FP Integer Without Inexact</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC0001C6</td>
<td>307</td>
<td>DFP drintq[] Decimal Floating Round To FP Integer With Inexact</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC0000C6</td>
<td>305</td>
<td>DFP drintx[] Decimal Floating Round To FP Integer With Inexact Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC0000C6</td>
<td>305</td>
<td>DFP drintnq[] Decimal Floating Round To FP Integer Without Inexact Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>59 EC000046</td>
<td>302</td>
<td>DFP drrnd[] Decimal Floating Reround</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z23</td>
<td>63 FC000046</td>
<td>302</td>
<td>DFP drrndq[] Decimal Floating Reround Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000604</td>
<td>310</td>
<td>DFP drsp[] Decimal Floating Round To DFP Short</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC000084</td>
<td>316</td>
<td>DFP dscl[] Decimal Floating Shift Coefficient Left Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC000084</td>
<td>316</td>
<td>DFP dsclq[] Decimal Floating Shift Coefficient Left Immediate Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC0000C4</td>
<td>316</td>
<td>DFP dscri[] Decimal Floating Shift Coefficient Right Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC0000C4</td>
<td>316</td>
<td>DFP dscriq[] Decimal Floating Shift Coefficient Right Immediate Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003C6</td>
<td>824</td>
<td>DS dsn Decorated Storage Notify</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000404</td>
<td>289</td>
<td>DFP dsur[] Decimal Floating Subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000404</td>
<td>289</td>
<td>DFP dsurq[] Decimal Floating Subtract Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC000184</td>
<td>298</td>
<td>DFP dstdc Decimal Floating Test Data Class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC000184</td>
<td>298</td>
<td>DFP dstdqc Decimal Floating Test Data Class Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>59 EC0001C4</td>
<td>296</td>
<td>DFP dstdgd Decimal Floating Test Data Group</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z22</td>
<td>63 FC0001C4</td>
<td>296</td>
<td>DFP dstdgdq Decimal Floating Test Data Group Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000144</td>
<td>297</td>
<td>DFP dtstex Decimal Floating Test Exponent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000144</td>
<td>297</td>
<td>DFP dtstexq Decimal Floating Test Exponent Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC000544</td>
<td>298</td>
<td>DFP dtstf Decimal Floating Test Significance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000544</td>
<td>298</td>
<td>DFP dtstfq Decimal Floating Test Significance Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC0002C4</td>
<td>314</td>
<td>DFP dxex[] Decimal Floating Extract Exponent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC0002C4</td>
<td>314</td>
<td>DFP dxexq[] Decimal Floating Extract Exponent Quad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00026C</td>
<td>826</td>
<td>EC eciw External Control In Word Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00036C</td>
<td>826</td>
<td>EC ecowx External Control Out Word Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Mnemonic Instruction</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------------------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E4</td>
<td>Floating-Point Double-Precision Absolute Value</td>
<td>SP.FD</td>
<td>efda</td>
<td>660</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E0</td>
<td>Floating-Point Double-Precision Add</td>
<td>SP.FD</td>
<td>eddd</td>
<td>661</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002EF</td>
<td>Convert Floating-Point Double-Precision from Single-Precision</td>
<td>SP.FD</td>
<td>efcf</td>
<td>666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F3</td>
<td>Convert Floating-Point Double-Precision from Signed Fraction</td>
<td>SP.FD</td>
<td>efcf</td>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F1</td>
<td>Convert Floating-Point Double-Precision from Signed Integer</td>
<td>SP.FD</td>
<td>efdos</td>
<td>663</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E3</td>
<td>Convert Floating-Point Double-Precision from Signed Integer Doubleword</td>
<td>SP.FD</td>
<td>efdcfu</td>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F2</td>
<td>Convert Floating-Point Double-Precision from Unsigned Fraction</td>
<td>SP.FD</td>
<td>efdcfu</td>
<td>663</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E2</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer</td>
<td>SP.FD</td>
<td>efdcfu</td>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002EE</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer Doubleword</td>
<td>SP.FD</td>
<td>efdcfu</td>
<td>662</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002EC</td>
<td>Floating-Point Double-Precision Compare Equal</td>
<td>SP.FD</td>
<td>efdcmp</td>
<td>662</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002ED</td>
<td>Floating-Point Double-Precision Compare Greater Than</td>
<td>SP.FD</td>
<td>edcmp</td>
<td>662</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F7</td>
<td>Convert Floating-Point Double-Precision to Signed Fraction</td>
<td>SP.FD</td>
<td>efdctsf</td>
<td>666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F5</td>
<td>Convert Floating-Point Double-Precision to Signed Integer</td>
<td>SP.FD</td>
<td>efdctsi</td>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002EB</td>
<td>Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero</td>
<td>SP.FD</td>
<td>efdctsidz</td>
<td>665</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002FA</td>
<td>Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero</td>
<td>SP.FD</td>
<td>efdctsz</td>
<td>666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F6</td>
<td>Convert Floating-Point Double-Precision to Unsigned Fraction</td>
<td>SP.FD</td>
<td>edcutf</td>
<td>666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F4</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer</td>
<td>SP.FD</td>
<td>edcutfi</td>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002EA</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero</td>
<td>SP.FD</td>
<td>edcutfidz</td>
<td>665</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002F8</td>
<td>Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero</td>
<td>SP.FD</td>
<td>edcutfui</td>
<td>666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E9</td>
<td>Floating-Point Double-Precision Divide</td>
<td>SP.FD</td>
<td>eddiv</td>
<td>661</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E8</td>
<td>Floating-Point Double-Precision Multiply</td>
<td>SP.FD</td>
<td>edfmul</td>
<td>661</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E5</td>
<td>Floating-Point Double-Precision Negative Absolute Value</td>
<td>SP.FD</td>
<td>efdrabs</td>
<td>660</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E6</td>
<td>Convert Floating-Point Double-Precision from Signed Fraction</td>
<td>SP.FD</td>
<td>efdsf</td>
<td>660</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002E1</td>
<td>Convert Floating-Point Double-Precision from Unsigned Integer</td>
<td>SP.FD</td>
<td>edsub</td>
<td>661</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002FE</td>
<td>Convert Floating-Point Double-Precision Test Equal</td>
<td>SP.FD</td>
<td>edtsteq</td>
<td>663</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002FC</td>
<td>Convert Floating-Point Double-Precision Test Greater Than</td>
<td>SP.FD</td>
<td>edtstgt</td>
<td>662</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002FD</td>
<td>Convert Floating-Point Double-Precision Test Less Than</td>
<td>SP.FD</td>
<td>edtstlt</td>
<td>663</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002C4</td>
<td>Floating-Point Absolute Value</td>
<td>SP.FS</td>
<td>efsabs</td>
<td>653</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002C0</td>
<td>Floating-Point Add</td>
<td>SP.FS</td>
<td>efsadd</td>
<td>654</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002CF</td>
<td>Convert Floating-Point Single-Precision Convert from Double-Precision</td>
<td>SP.FD</td>
<td>efsf</td>
<td>667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002D3</td>
<td>Convert Floating-Point from Signed Fraction</td>
<td>SP.FS</td>
<td>efsfcsf</td>
<td>658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX 4 100002D1</td>
<td>Convert Floating-Point from Signed Integer</td>
<td>SP.FS</td>
<td>efsfcsfi</td>
<td>658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Primary Base (hex)</td>
<td>Opcode Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D2</td>
<td>658  SP.FS</td>
<td>efscuf</td>
<td>Convert Floating-Point from Unsigned Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D0</td>
<td>658  SP.FS</td>
<td>efscufi</td>
<td>Convert Floating-Point from Unsigned Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002CE</td>
<td>658  SP.FS</td>
<td>efscmpgq</td>
<td>Floating-Point Compare Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002CC</td>
<td>655  SP.FS</td>
<td>efscmpgt</td>
<td>Floating-Point Compare Greater Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002CD</td>
<td>655  SP.FS</td>
<td>efscmplt</td>
<td>Floating-Point Compare Less Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D7</td>
<td>659  SP.FS</td>
<td>efscdfs</td>
<td>Convert Floating-Point to Signed Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D5</td>
<td>658  SP.FS</td>
<td>efscst</td>
<td>Convert Floating-Point to Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002DA</td>
<td>659  SP.FS</td>
<td>efscstsz</td>
<td>Convert Floating-Point to Signed Integer with Round toward Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D6</td>
<td>659  SP.FS</td>
<td>efscsf</td>
<td>Convert Floating-Point to Unsigned Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D4</td>
<td>658  SP.FS</td>
<td>efscui</td>
<td>Convert Floating-Point to Unsigned Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002D8</td>
<td>659  SP.FS</td>
<td>efscstuz</td>
<td>Convert Floating-Point to Unsigned Integer with Round toward Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002C9</td>
<td>654  SP.FS</td>
<td>efscdiv</td>
<td>Floating-Point Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002C8</td>
<td>654  SP.FS</td>
<td>efscmul</td>
<td>Floating-Point Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002C5</td>
<td>653  SP.FS</td>
<td>efscnsabs</td>
<td>Floating-Point Negative Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002C6</td>
<td>653  SP.FS</td>
<td>efscneg</td>
<td>Floating-Point Negate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002C1</td>
<td>654  SP.FS</td>
<td>efscsub</td>
<td>Floating-Point Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002DE</td>
<td>657  SP.FS</td>
<td>efststeq</td>
<td>Floating-Point Test Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002DC</td>
<td>656  SP.FS</td>
<td>efststgt</td>
<td>Floating-Point Test Greater Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002DD</td>
<td>657  SP.FS</td>
<td>efststlt</td>
<td>Floating-Point Test Less Than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>31 7C00021C</td>
<td>1043  E,HV</td>
<td>ehpriv</td>
<td>Embedded Hypervisor Privilege</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C00023B</td>
<td>85  S</td>
<td>eieio</td>
<td>Enforce In-order Execution of I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000208</td>
<td>594  SP</td>
<td>evabs</td>
<td>Vector Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000202</td>
<td>594  SP</td>
<td>evadd</td>
<td>Vector Add Immediate Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C9</td>
<td>594  SP</td>
<td>evaddmiaaw</td>
<td>Vector Add Signed, Modulo, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C1</td>
<td>595  SP</td>
<td>evaddssiaaw</td>
<td>Vector Add Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C8</td>
<td>595  SP</td>
<td>evaddumiaaw</td>
<td>Vector Add Unsigned, Modulo, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C0</td>
<td>595  SP</td>
<td>evadusiaaw</td>
<td>Vector Add Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000200</td>
<td>595  SP</td>
<td>evaddw</td>
<td>Vector Add Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000211</td>
<td>596  SP</td>
<td>evand</td>
<td>Vector AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000212</td>
<td>596  SP</td>
<td>evandc</td>
<td>Vector AND with Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000234</td>
<td>596  SP</td>
<td>evcmeq</td>
<td>Vector Compare Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000231</td>
<td>596  SP</td>
<td>evcmpgt</td>
<td>Vector Compare Greater Than Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000230</td>
<td>597  SP</td>
<td>evcmpgtu</td>
<td>Vector Compare Greater Than Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000233</td>
<td>597  SP</td>
<td>evcmps</td>
<td>Vector Compare Less Than Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000232</td>
<td>597  SP</td>
<td>evcmpsl</td>
<td>Vector Compare Less Than Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002BE</td>
<td>598  SP</td>
<td>evcntls</td>
<td>Vector Count Leading Signed Bits Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100002BD</td>
<td>598  SP</td>
<td>evcntlz</td>
<td>Vector Count Leading Zeros Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C6</td>
<td>598  SP</td>
<td>evdivws</td>
<td>Vector Divide Word Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C7</td>
<td>599  SP</td>
<td>evdivwu</td>
<td>Vector Divide Word Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000219</td>
<td>599  SP</td>
<td>eveqv</td>
<td>Vector Equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000020A</td>
<td>599  SP</td>
<td>evextsb</td>
<td>Vector Extend Sign Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000020B</td>
<td>599  SP</td>
<td>evextsh</td>
<td>Vector Extend Sign Half Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000284</td>
<td>645  SP.FV</td>
<td>evfsabs</td>
<td>Vector Floating-Point Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000280</td>
<td>646  SP.FV</td>
<td>evfsadd</td>
<td>Vector Floating-Point Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000293</td>
<td>650  SP.FV</td>
<td>evfsccsf</td>
<td>Vector Convert Floating-Point from Signed Fraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>------------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000291</td>
<td></td>
<td>650</td>
<td>SP.FV</td>
<td>evfscsi</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000292</td>
<td></td>
<td>650</td>
<td>SP.FV</td>
<td>evfscuf</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000290</td>
<td></td>
<td>650</td>
<td>SP.FV</td>
<td>evfscui</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000028E</td>
<td></td>
<td>648</td>
<td>SP.FV</td>
<td>evfscmpeq</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000028C</td>
<td></td>
<td>647</td>
<td>SP.FV</td>
<td>evfscmpgt</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000028D</td>
<td></td>
<td>647</td>
<td>SP.FV</td>
<td>evfscmplt</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000297</td>
<td></td>
<td>652</td>
<td>SP.FV</td>
<td>evfsctsf</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000295</td>
<td></td>
<td>651</td>
<td>SP.FV</td>
<td>evfsctsi</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029A</td>
<td></td>
<td>651</td>
<td>SP.FV</td>
<td>evfsctsz</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000296</td>
<td></td>
<td>652</td>
<td>SP.FV</td>
<td>evfsctuf</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000294</td>
<td></td>
<td>651</td>
<td>SP.FV</td>
<td>evfsctui</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000298</td>
<td></td>
<td>651</td>
<td>SP.FV</td>
<td>evfsctuiz</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000289</td>
<td></td>
<td>646</td>
<td>SP.FV</td>
<td>evfsdiv</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000288</td>
<td></td>
<td>646</td>
<td>SP.FV</td>
<td>evfsmul</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000285</td>
<td></td>
<td>645</td>
<td>SP.FV</td>
<td>evfnsabs</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000286</td>
<td></td>
<td>645</td>
<td>SP.FV</td>
<td>evfnsneg</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000281</td>
<td></td>
<td>646</td>
<td>SP.FV</td>
<td>evfssub</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029E</td>
<td></td>
<td>649</td>
<td>SP.FV</td>
<td>evfststeq</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029C</td>
<td></td>
<td>648</td>
<td>SP.FV</td>
<td>evfststgt</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000029D</td>
<td></td>
<td>649</td>
<td>SP.FV</td>
<td>evfststlt</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000301</td>
<td></td>
<td>600</td>
<td>SP</td>
<td>evladd</td>
</tr>
<tr>
<td>EVX</td>
<td>31</td>
<td>7C00063E</td>
<td>P</td>
<td>1069</td>
<td>E.PD</td>
<td>evldepx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000300</td>
<td></td>
<td>600</td>
<td>SP</td>
<td>evladdx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000305</td>
<td></td>
<td>600</td>
<td>SP</td>
<td>evldh</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000304</td>
<td></td>
<td>600</td>
<td>SP</td>
<td>evldhx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000303</td>
<td></td>
<td>601</td>
<td>SP</td>
<td>evldw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000302</td>
<td></td>
<td>601</td>
<td>SP</td>
<td>evldwx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000309</td>
<td></td>
<td>601</td>
<td>SP</td>
<td>evlhesplat</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000308</td>
<td></td>
<td>601</td>
<td>SP</td>
<td>evlhesplax</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030F</td>
<td></td>
<td>602</td>
<td>SP</td>
<td>evlhoossplat</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030E</td>
<td></td>
<td>602</td>
<td>SP</td>
<td>evlhoossplax</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030D</td>
<td></td>
<td>602</td>
<td>SP</td>
<td>evlhoussplat</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000030C</td>
<td></td>
<td>602</td>
<td>SP</td>
<td>evlhoussplax</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000311</td>
<td></td>
<td>603</td>
<td>SP</td>
<td>evlwhe</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000310</td>
<td></td>
<td>603</td>
<td>SP</td>
<td>evlwhex</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000317</td>
<td></td>
<td>603</td>
<td>SP</td>
<td>evlwhos</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000316</td>
<td>SP</td>
<td>603</td>
<td>SP</td>
<td>evlwhosx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000315</td>
<td>SP</td>
<td>604</td>
<td>SP</td>
<td>evlwhou</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000314</td>
<td>SP</td>
<td>604</td>
<td>SP</td>
<td>evlwhoux</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000031D</td>
<td>SP</td>
<td>604</td>
<td>SP</td>
<td>evlwhsplat</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000031C</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evlwhsplatx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000319</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evlwwsplat</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000318</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evlwwsplatx</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000322C</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evmergehi</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000322E</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evmergehilow</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000322D</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evmergecolo</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000322F</td>
<td>SP</td>
<td>605</td>
<td>SP</td>
<td>evmergecolohi</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052B</td>
<td>SP</td>
<td>606</td>
<td>SP</td>
<td>evmhegsmfaa</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000529</td>
<td>SP</td>
<td>607</td>
<td>SP</td>
<td>evmhegsmiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005A9</td>
<td>SP</td>
<td>607</td>
<td>SP</td>
<td>evmhegsmian</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000528</td>
<td>SP</td>
<td>607</td>
<td>SP</td>
<td>evmhegumiaa</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005A8</td>
<td>SP</td>
<td>607</td>
<td>SP</td>
<td>evmhegumian</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000040B</td>
<td>SP</td>
<td>608</td>
<td>SP</td>
<td>evmhesmf</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000042B</td>
<td>SP</td>
<td>608</td>
<td>SP</td>
<td>evmhesmfa</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000050B</td>
<td>SP</td>
<td>608</td>
<td>SP</td>
<td>evmhesmfaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000058B</td>
<td>SP</td>
<td>608</td>
<td>SP</td>
<td>evmhesmfanw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000409</td>
<td>SP</td>
<td>609</td>
<td>SP</td>
<td>evmhesmi</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000429</td>
<td>SP</td>
<td>609</td>
<td>SP</td>
<td>evmhesmia</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000509</td>
<td>SP</td>
<td>609</td>
<td>SP</td>
<td>evmhesmiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000589</td>
<td>SP</td>
<td>609</td>
<td>SP</td>
<td>evmhesmianw</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000403</td>
<td>SP</td>
<td>610</td>
<td>SP</td>
<td>evmhesssf</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000423</td>
<td>SP</td>
<td>610</td>
<td>SP</td>
<td>evmhesssfa</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000503</td>
<td>SP</td>
<td>611</td>
<td>SP</td>
<td>evmhesssfaaw</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Mnemonic</td>
<td>Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000583</td>
<td>611</td>
<td>SP</td>
<td>evmhessfanw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000501</td>
<td>612</td>
<td>SP</td>
<td>evmhessiaaw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000581</td>
<td>612</td>
<td>SP</td>
<td>evmhessianw</td>
<td>Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000408</td>
<td>613</td>
<td>SP</td>
<td>evmheumi</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000428</td>
<td>613</td>
<td>SP</td>
<td>evmheumia</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000508</td>
<td>613</td>
<td>SP</td>
<td>evmheumiaaw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000588</td>
<td>613</td>
<td>SP</td>
<td>evmheumianw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Modulo and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000500</td>
<td>614</td>
<td>SP</td>
<td>evmheusiaaw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Saturate and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000580</td>
<td>614</td>
<td>SP</td>
<td>evmheusianw</td>
<td>Vector Multiply Half Words, Even, Unsigned, Saturate and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052F</td>
<td>615</td>
<td>SP</td>
<td>evmhogsmfaa</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AF</td>
<td>615</td>
<td>SP</td>
<td>evmhogsmfan</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052D</td>
<td>615</td>
<td>SP</td>
<td>evmhogsmiaa</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AD</td>
<td>615</td>
<td>SP</td>
<td>evmhogsmian</td>
<td>Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000052C</td>
<td>616</td>
<td>SP</td>
<td>evmhogumiaa</td>
<td>Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>100005AC</td>
<td>616</td>
<td>SP</td>
<td>evmhogumian</td>
<td>Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000040F</td>
<td>616</td>
<td>SP</td>
<td>evmhosmf</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000042F</td>
<td>616</td>
<td>SP</td>
<td>evmhosmfa</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000050F</td>
<td>617</td>
<td>SP</td>
<td>evmhosmfaw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000058F</td>
<td>617</td>
<td>SP</td>
<td>evmhosmfanw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000040D</td>
<td>617</td>
<td>SP</td>
<td>evmhosmi</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000042D</td>
<td>617</td>
<td>SP</td>
<td>evmhosmia</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000050D</td>
<td>618</td>
<td>SP</td>
<td>evmhosmiaaw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>1000058D</td>
<td>617</td>
<td>SP</td>
<td>evmhosmianw</td>
<td>Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words</td>
</tr>
<tr>
<td>EVX</td>
<td>4</td>
<td>10000407</td>
<td>619</td>
<td>SP</td>
<td>evmhosss</td>
<td>Vector Multiply Half Words, Odd, Signed, Saturate, Fractional</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>10000427</td>
<td></td>
<td></td>
<td>619</td>
<td>SP</td>
<td>evmhossfa</td>
</tr>
<tr>
<td>EVX</td>
<td>10000507</td>
<td></td>
<td></td>
<td>620</td>
<td>SP</td>
<td>evmhossfaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000587</td>
<td></td>
<td></td>
<td>620</td>
<td>SP</td>
<td>evmhossfanw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000505</td>
<td></td>
<td></td>
<td>621</td>
<td>SP</td>
<td>evmhossiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000585</td>
<td></td>
<td></td>
<td>621</td>
<td>SP</td>
<td>evmhossianw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000040C</td>
<td></td>
<td></td>
<td>621</td>
<td>SP</td>
<td>evmhouri</td>
</tr>
<tr>
<td>EVX</td>
<td>1000042C</td>
<td></td>
<td></td>
<td>621</td>
<td>SP</td>
<td>evmhouriaw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000050C</td>
<td></td>
<td></td>
<td>622</td>
<td>SP</td>
<td>evmhouriaw</td>
</tr>
<tr>
<td>EVX</td>
<td>1000058C</td>
<td></td>
<td></td>
<td>618</td>
<td>SP</td>
<td>evmhouriaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000504</td>
<td></td>
<td></td>
<td>622</td>
<td>SP</td>
<td>evmhousiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000584</td>
<td></td>
<td></td>
<td>622</td>
<td>SP</td>
<td>evmhousianw</td>
</tr>
<tr>
<td>EVX</td>
<td>100004C4</td>
<td></td>
<td></td>
<td>623</td>
<td>SP</td>
<td>evmra</td>
</tr>
<tr>
<td>EVX</td>
<td>1000044F</td>
<td></td>
<td></td>
<td>623</td>
<td>SP</td>
<td>evmwhsmf</td>
</tr>
<tr>
<td>EVX</td>
<td>1000046F</td>
<td></td>
<td></td>
<td>623</td>
<td>SP</td>
<td>evmwhsfa</td>
</tr>
<tr>
<td>EVX</td>
<td>1000044D</td>
<td></td>
<td></td>
<td>623</td>
<td>SP</td>
<td>evmwhsni</td>
</tr>
<tr>
<td>EVX</td>
<td>1000046D</td>
<td></td>
<td></td>
<td>623</td>
<td>SP</td>
<td>evmwhsma</td>
</tr>
<tr>
<td>EVX</td>
<td>10000447</td>
<td></td>
<td></td>
<td>624</td>
<td>SP</td>
<td>evmwhssf</td>
</tr>
<tr>
<td>EVX</td>
<td>10000467</td>
<td></td>
<td></td>
<td>624</td>
<td>SP</td>
<td>evmwhssfa</td>
</tr>
<tr>
<td>EVX</td>
<td>1000044C</td>
<td></td>
<td></td>
<td>624</td>
<td>SP</td>
<td>evmwhumi</td>
</tr>
<tr>
<td>EVX</td>
<td>1000046C</td>
<td></td>
<td></td>
<td>624</td>
<td>SP</td>
<td>evmwhumia</td>
</tr>
<tr>
<td>EVX</td>
<td>10000549</td>
<td></td>
<td></td>
<td>625</td>
<td>SP</td>
<td>evmwlsmiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>100005C9</td>
<td></td>
<td></td>
<td>625</td>
<td>SP</td>
<td>evmwlsmianw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000541</td>
<td></td>
<td></td>
<td>625</td>
<td>SP</td>
<td>evmwlssiaaw</td>
</tr>
<tr>
<td>EVX</td>
<td>100005C1</td>
<td></td>
<td></td>
<td>625</td>
<td>SP</td>
<td>evmwlssianw</td>
</tr>
<tr>
<td>EVX</td>
<td>10000448</td>
<td></td>
<td></td>
<td>626</td>
<td>SP</td>
<td>evmwlumi</td>
</tr>
<tr>
<td>EVX</td>
<td>10000468</td>
<td></td>
<td></td>
<td>626</td>
<td>SP</td>
<td>evmwlumia</td>
</tr>
<tr>
<td>EVX</td>
<td>10000548</td>
<td></td>
<td></td>
<td>626</td>
<td>SP</td>
<td>evmwlumiaaw</td>
</tr>
<tr>
<td>Format</td>
<td>Primary Base (hex)</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>100005C8</td>
<td>626</td>
<td>SP</td>
<td>evmwlumianw</td>
<td>Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000540</td>
<td>627</td>
<td>SP</td>
<td>evmwlusiaaw</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>100005C0</td>
<td>627</td>
<td>SP</td>
<td>evmwlusianw</td>
<td>Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000045B</td>
<td>627</td>
<td>SP</td>
<td>evmwsmf</td>
<td>Vector Multiply Word Signed, Modulo, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000047B</td>
<td>627</td>
<td>SP</td>
<td>evmwsmfa</td>
<td>Vector Multiply Word Signed, Modulo, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000055B</td>
<td>628</td>
<td>SP</td>
<td>evmwsmfaa</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>100005DB</td>
<td>628</td>
<td>SP</td>
<td>evmwsmfan</td>
<td>Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000459</td>
<td>628</td>
<td>SP</td>
<td>evmwsmi</td>
<td>Vector Multiply Word Signed, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000479</td>
<td>628</td>
<td>SP</td>
<td>evmwsmia</td>
<td>Vector Multiply Word Signed, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000559</td>
<td>628</td>
<td>SP</td>
<td>evmwsmiia</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>100005D9</td>
<td>628</td>
<td>SP</td>
<td>evmwsmian</td>
<td>Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000453</td>
<td>629</td>
<td>SP</td>
<td>evmwssf</td>
<td>Vector Multiply Word Signed, Saturate, Fractional</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000473</td>
<td>629</td>
<td>SP</td>
<td>evmwssfa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000553</td>
<td>629</td>
<td>SP</td>
<td>evmwssfaa</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>100005D3</td>
<td>630</td>
<td>SP</td>
<td>evmwssfan</td>
<td>Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000458</td>
<td>630</td>
<td>SP</td>
<td>evmwumi</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000478</td>
<td>630</td>
<td>SP</td>
<td>evmwumia</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer to Accumulator</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000558</td>
<td>631</td>
<td>SP</td>
<td>evmwumiaa</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer and Accumulate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>100005D8</td>
<td>631</td>
<td>SP</td>
<td>evmwumian</td>
<td>Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000021E</td>
<td>631</td>
<td>SP</td>
<td>evnand</td>
<td>Vector NAND</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000209</td>
<td>631</td>
<td>SP</td>
<td>evneg</td>
<td>Vector Negate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000218</td>
<td>631</td>
<td>SP</td>
<td>evnor</td>
<td>Vector NOR</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000217</td>
<td>632</td>
<td>SP</td>
<td>evor</td>
<td>Vector OR</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000021B</td>
<td>632</td>
<td>SP</td>
<td>evorc</td>
<td>Vector OR with Complement</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000228</td>
<td>632</td>
<td>SP</td>
<td>evrlw</td>
<td>Vector Rotate Left Word</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000022A</td>
<td>633</td>
<td>SP</td>
<td>evrlwi</td>
<td>Vector Rotate Left Word Immediate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000020C</td>
<td>633</td>
<td>SP</td>
<td>evrdw</td>
<td>Vector Round Word</td>
<td></td>
</tr>
<tr>
<td>EVS 4</td>
<td>10000278</td>
<td>633</td>
<td>SP</td>
<td>evsel</td>
<td>Vector Select</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000224</td>
<td>634</td>
<td>SP</td>
<td>evslw</td>
<td>Vector Shift Left Word</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000226</td>
<td>634</td>
<td>SP</td>
<td>evslwi</td>
<td>Vector Shift Left Word Immediate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>1000022B</td>
<td>634</td>
<td>SP</td>
<td>evsplati</td>
<td>Vector Splat Fractional Immediate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000229</td>
<td>634</td>
<td>SP</td>
<td>evsplati</td>
<td>Vector Splat Immediate</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000223</td>
<td>634</td>
<td>SP</td>
<td>evsrwls</td>
<td>Vector Shift Right Word Immediate Signed</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000222</td>
<td>634</td>
<td>SP</td>
<td>evsrwlsu</td>
<td>Vector Shift Right Word Immediate Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000221</td>
<td>635</td>
<td>SP</td>
<td>evsrsw</td>
<td>Vector Shift Right Word Signed</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000220</td>
<td>635</td>
<td>SP</td>
<td>evsrwu</td>
<td>Vector Shift Right Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>EVX 4</td>
<td>10000321</td>
<td>635</td>
<td>SP</td>
<td>evstdd</td>
<td>Vector Store Double of Double</td>
<td></td>
</tr>
<tr>
<td>EVX 31</td>
<td>7C00073E</td>
<td>P 1069</td>
<td>E.PD</td>
<td>evstddepx</td>
<td>Vector Store Double of Double by External PID Indexed</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000320</td>
<td>635</td>
<td>SP</td>
<td>evstddx</td>
<td>Vector Store Double of Double Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000325</td>
<td>636</td>
<td>SP</td>
<td>evstdh</td>
<td>Vector Store Double of Four Half Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000324</td>
<td>636</td>
<td>SP</td>
<td>evstdhx</td>
<td>Vector Store Double of Four Half Words Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000323</td>
<td>636</td>
<td>SP</td>
<td>evstdw</td>
<td>Vector Store Double of Two Words</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000322</td>
<td>636</td>
<td>SP</td>
<td>evstdwx</td>
<td>Vector Store Double of Two Words Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000331</td>
<td>637</td>
<td>SP</td>
<td>evstwhex</td>
<td>Vector Store Word of Two Half Words from Even Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000335</td>
<td>637</td>
<td>SP</td>
<td>evstwho</td>
<td>Vector Store Word of Two Half Words from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000334</td>
<td>637</td>
<td>SP</td>
<td>evstwhox</td>
<td>Vector Store Word of Two Half Words from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000339</td>
<td>637</td>
<td>SP</td>
<td>evstww</td>
<td>Vector Store Word of Word from Even</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000338</td>
<td>637</td>
<td>SP</td>
<td>evstwwex</td>
<td>Vector Store Word of Word from Even Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000033D</td>
<td>638</td>
<td>SP</td>
<td>evstwox</td>
<td>Vector Store Word of Word from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000033C</td>
<td>638</td>
<td>SP</td>
<td>evstwwo</td>
<td>Vector Store Word of Word from Odd</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 1000033F</td>
<td>638</td>
<td>SP</td>
<td>evstwwx</td>
<td>Vector Store Word of Word from Odd Indexed</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004CB</td>
<td>638</td>
<td>SP</td>
<td>evsubfsiaaw</td>
<td>Vector Subtract Signed, Modulo, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C3</td>
<td>638</td>
<td>SP</td>
<td>evsubfssiaaw</td>
<td>Vector Subtract Signed, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004CA</td>
<td>639</td>
<td>SP</td>
<td>evsubfumiaaw</td>
<td>Vector Subtract Unsigned, Modulo, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 100004C2</td>
<td>639</td>
<td>SP</td>
<td>evsubfusiaaw</td>
<td>Vector Subtract Unsigned, Saturate, Integer to Accumulator Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000204</td>
<td>639</td>
<td>SP</td>
<td>evsubf</td>
<td>Vector Subtract from Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000206</td>
<td>639</td>
<td>SP</td>
<td>evsubfw</td>
<td>Vector Subtract Immediate from Word</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>4 10000216</td>
<td>639</td>
<td>SP</td>
<td>evxor</td>
<td>Vector XOR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000774</td>
<td>85</td>
<td>B</td>
<td>extsb[]</td>
<td>Extend Sign Byte</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000734</td>
<td>85</td>
<td>B</td>
<td>extsh[]</td>
<td>Extend Sign Halfword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007B4</td>
<td>89</td>
<td>64</td>
<td>extsw[]</td>
<td>Extend Sign Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000210</td>
<td>142</td>
<td>FP[R]</td>
<td>fabs[]</td>
<td>Floating Absolute Value</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00002A</td>
<td>144</td>
<td>FP[R]</td>
<td>fadd[]</td>
<td>Floating Add</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC00002A</td>
<td>144</td>
<td>FP[R]</td>
<td>fadd[]</td>
<td>Floating Add Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00069C</td>
<td>155</td>
<td>FP[R]</td>
<td>fcfid[]</td>
<td>Floating Convert From Integer Doubleword Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC00069C</td>
<td>156</td>
<td>FP[R]</td>
<td>fcfids[]</td>
<td>Floating Convert From Integer Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC00079C</td>
<td>156</td>
<td>FP[R]</td>
<td>fcfidu[]</td>
<td>Floating Convert From Integer Doubleword Unsigned</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>59 EC00079C</td>
<td>157</td>
<td>FP[R]</td>
<td>fcfidu[]</td>
<td>Floating Convert From Integer Doubleword Unsigned Single</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000040</td>
<td>159</td>
<td>FP</td>
<td>fcmpo</td>
<td>Floating Compare Ordered</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000000</td>
<td>159</td>
<td>FP</td>
<td>fcmpu</td>
<td>Floating Compare Unordered</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000010</td>
<td>142</td>
<td>FP[R]</td>
<td>fcpsgn[]</td>
<td>Floating Copy Sign</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000065C</td>
<td>151</td>
<td>FP[R]</td>
<td>fctid[]</td>
<td>Floating Convert To Integer Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000075C</td>
<td>152</td>
<td>FP[R]</td>
<td>fctidu[]</td>
<td>Floating Convert To Integer Doubleword Unsigned</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000075E</td>
<td>153</td>
<td>FP[R]</td>
<td>fctiduz[]</td>
<td>Floating Convert To Integer Doubleword Unsigned with round toward Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000065E</td>
<td>152</td>
<td>FP[R]</td>
<td>fctidz[]</td>
<td>Floating Convert To Integer Doubleword with round toward Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000001C</td>
<td>153</td>
<td>FP[R]</td>
<td>fctiw[]</td>
<td>Floating Convert To Integer Word</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000011C</td>
<td>154</td>
<td>FP[R]</td>
<td>fctiwu[]</td>
<td>Floating Convert To Integer Word Unsigned</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>63 FC000011B</td>
<td>155</td>
<td>FP[R]</td>
<td>fctiwu[]</td>
<td>Floating Convert To Integer Word Unsigned with round toward Zero</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Page</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC00001E</td>
<td>154</td>
<td>FP[R]</td>
<td>fctiwz[.]</td>
<td>Floating Convert To Integer Word with round to Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC000024</td>
<td>145</td>
<td>FP[R]</td>
<td>tdiv[.]</td>
<td>Floating Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC000024</td>
<td>145</td>
<td>FP[R]</td>
<td>tdivs[.]</td>
<td>Floating Divide Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC00003A</td>
<td>149</td>
<td>FP[R]</td>
<td>fmad[.]</td>
<td>Floating Multiply-Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC00003A</td>
<td>149</td>
<td>FP[R]</td>
<td>fmadss[.]</td>
<td>Floating Multiply-Add Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000090</td>
<td>142</td>
<td>FP[R]</td>
<td>frm[.]</td>
<td>Floating Move Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC00078C</td>
<td>143</td>
<td>VSX</td>
<td>fmgew</td>
<td>Floating Merge Even Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC00068C</td>
<td>143</td>
<td>VSX</td>
<td>fmgow</td>
<td>Floating Merge Odd Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC000038</td>
<td>149</td>
<td>FP[R]</td>
<td>fmsub[.]</td>
<td>Floating Multiply-Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC000038</td>
<td>149</td>
<td>FP[R]</td>
<td>fmsubs[.]</td>
<td>Floating Multiply-Subtract Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC000032</td>
<td>145</td>
<td>FP[R]</td>
<td>fmul[.]</td>
<td>Floating Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC000032</td>
<td>145</td>
<td>FP[R]</td>
<td>fmuls[.]</td>
<td>Floating Multiply Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000110</td>
<td>142</td>
<td>FP[R]</td>
<td>fnabs[.]</td>
<td>Floating Negative Absolute Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000050</td>
<td>142</td>
<td>FP[R]</td>
<td>fneg[.]</td>
<td>Floating Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC00003E</td>
<td>150</td>
<td>FP[R]</td>
<td>fnmad[.]</td>
<td>Floating Negative Multiply-Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC00003E</td>
<td>150</td>
<td>FP[R]</td>
<td>fnmadss[.]</td>
<td>Floating Negative Multiply-Add Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC00003C</td>
<td>150</td>
<td>FP[R]</td>
<td>fnmsub[.]</td>
<td>Floating Negative Multiply-Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC00003C</td>
<td>150</td>
<td>FP[R]</td>
<td>fnmsubs[.]</td>
<td>Floating Negative Multiply-Subtract Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC000030</td>
<td>146</td>
<td>FP[R].i</td>
<td>ffre[.]</td>
<td>Floating Reciprocal Estimate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC000030</td>
<td>146</td>
<td>FP[R].i</td>
<td>fres[.]</td>
<td>Floating Reciprocal Estimate Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC0003D0</td>
<td>158</td>
<td>FP[R].i</td>
<td>ffr[.]</td>
<td>Floating Round To Integer Minus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000310</td>
<td>158</td>
<td>FP[R].i</td>
<td>frin[.]</td>
<td>Floating Round To Integer Nearest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000390</td>
<td>158</td>
<td>FP[R].i</td>
<td>fnp[.]</td>
<td>Floating Round To Integer Plus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000350</td>
<td>158</td>
<td>FP[R].i</td>
<td>frz[.]</td>
<td>Floating Round To Integer toward Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000018</td>
<td>151</td>
<td>FP[R]</td>
<td>frp[.]</td>
<td>Floating Round to Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC000034</td>
<td>147</td>
<td>FP[R]</td>
<td>frsqte[.]</td>
<td>Floating Reciprocal Square Root Estimate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC000034</td>
<td>147</td>
<td>FP[R].i</td>
<td>frsqrtes[.]</td>
<td>Floating Reciprocal Square Root Estimate Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC00002E</td>
<td>160</td>
<td>FP[R]</td>
<td>fse[.]</td>
<td>Floating Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC00002C</td>
<td>146</td>
<td>FP[R]</td>
<td>fso[.]</td>
<td>Floating Square Root</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC00002C</td>
<td>146</td>
<td>FP[R]</td>
<td>fso[.]</td>
<td>Floating Square Root Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 63 FC000028</td>
<td>144</td>
<td>FP[R]</td>
<td>fsub[.]</td>
<td>Floating Subtract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 59 EC000028</td>
<td>144</td>
<td>FP[R]</td>
<td>fsubs[.]</td>
<td>Floating Subtract Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000100</td>
<td>148</td>
<td>FP</td>
<td>tdiv</td>
<td>Floating Test for software Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 63 FC000140</td>
<td>148</td>
<td>FP</td>
<td>tdiv</td>
<td>Floating Test for software Square Root</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL 19 4C000224</td>
<td>865</td>
<td>S</td>
<td>hrlfd</td>
<td>Return From Interrupt Doubleword Hypervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0007AC</td>
<td>762</td>
<td>B</td>
<td>icbi</td>
<td>Instruction Cache Block Invalidate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0007BE</td>
<td>1067</td>
<td>E.PD</td>
<td>icbiep</td>
<td>Instruction Cache Block Invalidate by External PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0011CC</td>
<td>1124</td>
<td>ECL</td>
<td>icblc</td>
<td>Instruction Cache Block Lock Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0011BD</td>
<td>1121</td>
<td>ECL</td>
<td>icbqlq</td>
<td>Instruction Cache Block Lock Query</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C00012C</td>
<td>762</td>
<td>E</td>
<td>icbt</td>
<td>Instruction Cache Block Touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0003CC</td>
<td>1123</td>
<td>ECL</td>
<td>icblls</td>
<td>Instruction Cache Block Touch and Lock Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0007BC</td>
<td>1239</td>
<td>E.CI</td>
<td>ici</td>
<td>Instruction Cache Invalidate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0007CC</td>
<td>1243</td>
<td>E.CD</td>
<td>icread</td>
<td>Instruction Cache Read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 31 7C0001E</td>
<td>81</td>
<td>B</td>
<td>isel</td>
<td>Integer Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL 19 4C00012C</td>
<td>776</td>
<td>B</td>
<td>isnc</td>
<td>Instruction Synchronize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C000068</td>
<td>777</td>
<td>B</td>
<td>lbax</td>
<td>Load Byte And Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C000406</td>
<td>822</td>
<td>DS</td>
<td>ldbx</td>
<td>Load Byte with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C000BE</td>
<td>1059</td>
<td>E.PD</td>
<td>lbepx</td>
<td>Load Byte and Zero by External PID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 34 88000000</td>
<td>48</td>
<td>B</td>
<td>lbz</td>
<td>Load Byte and Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31 7C0006AA</td>
<td>876</td>
<td>S</td>
<td>lbzccx</td>
<td>Load Byte and Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>D</td>
<td>C8000000</td>
<td>35</td>
<td>B</td>
<td>48</td>
<td></td>
<td>lbzu</td>
</tr>
<tr>
<td>X</td>
<td>7C0000EE</td>
<td>48</td>
<td>B</td>
<td>48</td>
<td></td>
<td>lbzux</td>
</tr>
<tr>
<td>X</td>
<td>7C0000AE</td>
<td>49</td>
<td>B</td>
<td>49</td>
<td></td>
<td>lbxz</td>
</tr>
<tr>
<td>DS</td>
<td>E8000000</td>
<td>53</td>
<td>64</td>
<td>53</td>
<td></td>
<td>ld</td>
</tr>
<tr>
<td>X</td>
<td>7C0000A8</td>
<td>782</td>
<td>64</td>
<td>782</td>
<td></td>
<td>ldarx</td>
</tr>
<tr>
<td>X</td>
<td>7C0000A8</td>
<td>60</td>
<td>64</td>
<td>60</td>
<td></td>
<td>ldbrx</td>
</tr>
<tr>
<td>X</td>
<td>7C0000EA</td>
<td>H 876</td>
<td>S</td>
<td>876</td>
<td></td>
<td>ldcix</td>
</tr>
<tr>
<td>X</td>
<td>7C0000C6</td>
<td>822</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>ldxx</td>
</tr>
<tr>
<td>X</td>
<td>7C00003A</td>
<td>P 1060</td>
<td>E.PD:64</td>
<td>1060</td>
<td></td>
<td>ldex</td>
</tr>
<tr>
<td>DS</td>
<td>E8000000</td>
<td>53</td>
<td>64</td>
<td>53</td>
<td></td>
<td>ldu</td>
</tr>
<tr>
<td>X</td>
<td>7C00006A</td>
<td>53</td>
<td>64</td>
<td>53</td>
<td></td>
<td>ldx</td>
</tr>
<tr>
<td>D</td>
<td>C8000000</td>
<td>135</td>
<td>FP</td>
<td>135</td>
<td></td>
<td>lfd</td>
</tr>
<tr>
<td>X</td>
<td>7C000046</td>
<td>822</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>ldxx</td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>P 1068</td>
<td>E.PD</td>
<td>1068</td>
<td></td>
<td>ldex</td>
</tr>
<tr>
<td>DS</td>
<td>E4000000</td>
<td>141</td>
<td>FP.out</td>
<td>141</td>
<td></td>
<td>lfdp</td>
</tr>
<tr>
<td>X</td>
<td>7C00006E</td>
<td>141</td>
<td>FP.out</td>
<td>141</td>
<td></td>
<td>lfpix</td>
</tr>
<tr>
<td>D</td>
<td>C8000000</td>
<td>135</td>
<td>FP</td>
<td>135</td>
<td></td>
<td>lfdx</td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>135</td>
<td>FP</td>
<td>135</td>
<td></td>
<td>lfdx</td>
</tr>
<tr>
<td>X</td>
<td>7C00006E</td>
<td>136</td>
<td>FP</td>
<td>136</td>
<td></td>
<td>lfidx</td>
</tr>
<tr>
<td>X</td>
<td>7C00006E</td>
<td>136</td>
<td>FP</td>
<td>136</td>
<td></td>
<td>lfidx</td>
</tr>
<tr>
<td>D</td>
<td>C0000000</td>
<td>138</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>lfs</td>
</tr>
<tr>
<td>D</td>
<td>C4000000</td>
<td>138</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>lfsu</td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>138</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>lfsx</td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>138</td>
<td>FP</td>
<td>138</td>
<td></td>
<td>lfsx</td>
</tr>
<tr>
<td>D</td>
<td>A8000000</td>
<td>50</td>
<td>B</td>
<td>50</td>
<td></td>
<td>lh</td>
</tr>
<tr>
<td>X</td>
<td>7C0000E8</td>
<td>778</td>
<td>B</td>
<td>778</td>
<td></td>
<td>lhax</td>
</tr>
<tr>
<td>D</td>
<td>A0000000</td>
<td>50</td>
<td>B</td>
<td>50</td>
<td></td>
<td>lhux</td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>50</td>
<td>B</td>
<td>50</td>
<td></td>
<td>lhux</td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>50</td>
<td>B</td>
<td>50</td>
<td></td>
<td>lhux</td>
</tr>
<tr>
<td>X</td>
<td>7C00002C</td>
<td>59</td>
<td>B</td>
<td>59</td>
<td></td>
<td>lhbrx</td>
</tr>
<tr>
<td>X</td>
<td>7C000044</td>
<td>822</td>
<td>DS</td>
<td>822</td>
<td></td>
<td>lhdx</td>
</tr>
<tr>
<td>X</td>
<td>7C00002B</td>
<td>P 1059</td>
<td>E.PD</td>
<td>1059</td>
<td></td>
<td>lhdx</td>
</tr>
<tr>
<td>D</td>
<td>A0000000</td>
<td>49</td>
<td>B</td>
<td>49</td>
<td></td>
<td>lhz</td>
</tr>
<tr>
<td>X</td>
<td>7C00006A</td>
<td>H 876</td>
<td>S</td>
<td>876</td>
<td></td>
<td>lhzcix</td>
</tr>
<tr>
<td>D</td>
<td>A4000000</td>
<td>49</td>
<td>B</td>
<td>49</td>
<td></td>
<td>lhzu</td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>49</td>
<td>B</td>
<td>49</td>
<td></td>
<td>lhzu</td>
</tr>
<tr>
<td>X</td>
<td>7C00002E</td>
<td>49</td>
<td>B</td>
<td>49</td>
<td></td>
<td>lhzu</td>
</tr>
<tr>
<td>D</td>
<td>B8000000</td>
<td>61</td>
<td>B</td>
<td>61</td>
<td></td>
<td>lmuw</td>
</tr>
<tr>
<td>DQ</td>
<td>E0000000</td>
<td>P 58</td>
<td>LSQ</td>
<td>58</td>
<td></td>
<td>lq</td>
</tr>
<tr>
<td>X</td>
<td>7C000028</td>
<td>784</td>
<td>LSQ</td>
<td>784</td>
<td></td>
<td>lqarx</td>
</tr>
<tr>
<td>X</td>
<td>7C00004A</td>
<td>63</td>
<td>MA</td>
<td>63</td>
<td></td>
<td>lsxi</td>
</tr>
<tr>
<td>X</td>
<td>7C00004A</td>
<td>63</td>
<td>MA</td>
<td>63</td>
<td></td>
<td>lsxw</td>
</tr>
<tr>
<td>X</td>
<td>7C00000E</td>
<td>184</td>
<td>V</td>
<td>184</td>
<td></td>
<td>lvebx</td>
</tr>
<tr>
<td>X</td>
<td>7C00004E</td>
<td>181</td>
<td>V</td>
<td>181</td>
<td></td>
<td>lvex</td>
</tr>
<tr>
<td>X</td>
<td>7C00002A</td>
<td>P 1070</td>
<td>E.PD</td>
<td>1070</td>
<td></td>
<td>lvapx</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.¹</td>
<td>Privilege¹</td>
<td>Page</td>
<td>Category¹</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>X  31 7C00020E</td>
<td>P 1070</td>
<td>E.PD</td>
<td>lvepxl</td>
<td>Load Vector by External PID Indexed Last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00008E</td>
<td>181</td>
<td>V</td>
<td>lvewx</td>
<td>Load Vector Element Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00004C</td>
<td>186</td>
<td>V</td>
<td>lvsl</td>
<td>Load Vector for Shift Left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00004C</td>
<td>186</td>
<td>V</td>
<td>lvsr</td>
<td>Load Vector for Shift Right</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C0002CE</td>
<td>182</td>
<td>V</td>
<td>lvxl</td>
<td>Load Vector Indexed Last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00008E</td>
<td>181</td>
<td>V</td>
<td>lvewx</td>
<td>Load Vector Element Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C0000CE</td>
<td>186</td>
<td>V</td>
<td>lvsl</td>
<td>Load Vector for Shift Left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C0000CE</td>
<td>186</td>
<td>V</td>
<td>lvsr</td>
<td>Load Vector for Shift Right</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C0002CE</td>
<td>182</td>
<td>V</td>
<td>lvxl</td>
<td>Load Vector Indexed Last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS  58 E8000002</td>
<td>52</td>
<td>64</td>
<td>lwa</td>
<td>Load Word Algebraic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00028</td>
<td>777</td>
<td>B</td>
<td>lwarx</td>
<td>Load Word and Reserve Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C0002EA</td>
<td>52</td>
<td>64</td>
<td>lwaux</td>
<td>Load Word Algebraic with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C0002AA</td>
<td>52</td>
<td>64</td>
<td>lwa</td>
<td>Load Word Algebraic Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00042C</td>
<td>59</td>
<td>B</td>
<td>lbbrx</td>
<td>Load Word Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C000486</td>
<td>822</td>
<td>DS</td>
<td>lwdx</td>
<td>Load Word with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00003E</td>
<td>P 1060</td>
<td>E.PD</td>
<td>lwepx</td>
<td>Load Word and Zero by External PID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D  32 80000000</td>
<td>51</td>
<td>B</td>
<td>lwz</td>
<td>Load Word and Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00062A</td>
<td>H 876</td>
<td>S</td>
<td>lwzcx</td>
<td>Load Word and Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D  33 84000000</td>
<td>51</td>
<td>B</td>
<td>lwzu</td>
<td>Load Word and Zero with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00062A</td>
<td>H 876</td>
<td>S</td>
<td>lwzcx</td>
<td>Load Word and Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X  31 7C00002E</td>
<td>51</td>
<td>B</td>
<td>lwz</td>
<td>Load Word and Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C000498</td>
<td>393</td>
<td>VSX</td>
<td>lxsdx</td>
<td>Load VSR Scalar Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C000098</td>
<td>393</td>
<td>VSX</td>
<td>lxsiwax</td>
<td>Load VSR Scalar as Integer Word Algebraic Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C00018</td>
<td>394</td>
<td>VSX</td>
<td>lxsiwzx</td>
<td>Load VSR Scalar as Integer Word and Zero Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C000418</td>
<td>394</td>
<td>VSX</td>
<td>lxsspx</td>
<td>Load VSR Scalar Single-Precision Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C000698</td>
<td>395</td>
<td>VSX</td>
<td>lxvd2x</td>
<td>Load VSR Vector Doubleword*2 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C000298</td>
<td>395</td>
<td>VSX</td>
<td>lxvd4x</td>
<td>Load VSR Vector Doubleword &amp; Splat Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX1 31 7C000618</td>
<td>396</td>
<td>VSX</td>
<td>lxvw4x</td>
<td>Load VSR Vector Word*4 Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000158</td>
<td>675</td>
<td>LMA</td>
<td>macchw[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Module Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000158</td>
<td>675</td>
<td>LMA</td>
<td>macchwo[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Module Signed &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 100001D8</td>
<td>675</td>
<td>LMA</td>
<td>macchws[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 100001D8</td>
<td>675</td>
<td>LMA</td>
<td>macchwso[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Signed &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000198</td>
<td>676</td>
<td>LMA</td>
<td>macchwsu[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000198</td>
<td>676</td>
<td>LMA</td>
<td>macchwsuo[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Saturate Unsigned &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000118</td>
<td>676</td>
<td>LMA</td>
<td>macchwu[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Modulo Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000118</td>
<td>676</td>
<td>LMA</td>
<td>macchwoo[.]</td>
<td>Multiply Accumulate Cross Halfword to Word Modulo Unsigned &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000058</td>
<td>677</td>
<td>LMA</td>
<td>machhw[.]</td>
<td>Multiply Accumulate High Halfword to Word Module Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000058</td>
<td>677</td>
<td>LMA</td>
<td>machhwo[.]</td>
<td>Multiply Accumulate High Halfword to Word Module Signed &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 100000D8</td>
<td>677</td>
<td>LMA</td>
<td>machhws[.]</td>
<td>Multiply Accumulate High Halfword to Word Saturate Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 100000D8</td>
<td>677</td>
<td>LMA</td>
<td>machhwsq[.]</td>
<td>Multiply Accumulate High Halfword to Word Saturate Signed &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000098</td>
<td>678</td>
<td>LMA</td>
<td>machhwsu[.]</td>
<td>Multiply Accumulate High Halfword to Word Saturate Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XO  4 10000098</td>
<td>678</td>
<td>LMA</td>
<td>machhwsuo[.]</td>
<td>Multiply Accumulate High Halfword to Word Saturate Unsigned &amp; record OV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>XO</td>
<td>10000018</td>
<td>678</td>
<td>LMA</td>
<td>machhwu[.]</td>
<td>Multiply Accumulate High Halfword to Word Modulo Unsigned</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000018</td>
<td>678</td>
<td>LMA</td>
<td>machhwu[.]</td>
<td>Multiply Accumulate High Halfword to Word Modulo Unsigned &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000358</td>
<td>679</td>
<td>LMA</td>
<td>macihw[.]</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Signed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000358</td>
<td>679</td>
<td>LMA</td>
<td>macihwo[.]</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>100003D8</td>
<td>679</td>
<td>LMA</td>
<td>macihws[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>100003D8</td>
<td>679</td>
<td>LMA</td>
<td>macihws[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000398</td>
<td>680</td>
<td>LMA</td>
<td>macihwsuo[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000398</td>
<td>680</td>
<td>LMA</td>
<td>macihwsuo[.]</td>
<td>Multiply Accumulate Low Halfword to Word Saturate Unsigned &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000318</td>
<td>680</td>
<td>LMA</td>
<td>macihw[.]</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Unsigned</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>10000318</td>
<td>680</td>
<td>LMA</td>
<td>macihw[.]</td>
<td>Multiply Accumulate Low Halfword to Word Modulo Unsigned &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0006AC</td>
<td>790</td>
<td>E</td>
<td>mbar</td>
<td>Memory Barrier</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>4C000000</td>
<td>42</td>
<td>B</td>
<td>mcrf</td>
<td>Move Condition Register Field</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FC000800</td>
<td>161</td>
<td>FP</td>
<td>mcrfs</td>
<td>Move To Condition Register from FPSCR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000400</td>
<td>113</td>
<td>E</td>
<td>mcrxr</td>
<td>Move to Condition Register from XER</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C00025C</td>
<td>44</td>
<td>S</td>
<td>mfbhrbe</td>
<td>Move From Branch History Rolling Buffer</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C00026E</td>
<td>108</td>
<td>B</td>
<td>mfr</td>
<td>Move From Condition Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000286</td>
<td>1055</td>
<td>E.DC</td>
<td>mfdcr</td>
<td>Move From Device Control Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000246</td>
<td>113</td>
<td>E.DC</td>
<td>mfdrux</td>
<td>Move From Device Control Register User Mode Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000206</td>
<td>1055</td>
<td>E.DC</td>
<td>mfdcrx</td>
<td>Move From Device Control Register Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>FC0004BE</td>
<td>161</td>
<td>FP[R]</td>
<td>mfs[,.]</td>
<td>Move From FPSCR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0000A6</td>
<td>888</td>
<td>S</td>
<td>mfsr</td>
<td>Move From Machine State Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00026E</td>
<td>112</td>
<td>B</td>
<td>mfcrcf</td>
<td>Move From One Condition Register Field</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C00029C</td>
<td>1257</td>
<td>E.PM</td>
<td>mfpmr</td>
<td>Move From Performance Monitor Register</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C0002A6</td>
<td>106</td>
<td>B</td>
<td>mfspr</td>
<td>Move From Special Purpose Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0004A6</td>
<td>927</td>
<td>S</td>
<td>mfsr</td>
<td>Move From Segment Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000526</td>
<td>927</td>
<td>S</td>
<td>mfsrin</td>
<td>Move From Segment Register Indirect</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C0002E6</td>
<td>813</td>
<td>S.out</td>
<td>mftb</td>
<td>Move From Time Base</td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>10000604</td>
<td>268</td>
<td>V</td>
<td>mvfscr</td>
<td>Move From Vector Status and Control Register</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>7C000066</td>
<td>109</td>
<td>VSX</td>
<td>mfrsd</td>
<td>Move From VSR Doubleword</td>
<td></td>
</tr>
<tr>
<td>XX1</td>
<td>7C0000E6</td>
<td>109</td>
<td>VSX</td>
<td>mfrswrzw</td>
<td>Move From VSR Word and Zero</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C0001DC</td>
<td>1006</td>
<td>S.PC</td>
<td>msgclr</td>
<td>Message Clear</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00015C</td>
<td>1007</td>
<td>S.PC</td>
<td>msgclrp</td>
<td>Message Clear Privileged</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00019C</td>
<td>1006</td>
<td>S.PC</td>
<td>msgsnd</td>
<td>Message Send</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C00011C</td>
<td>1007</td>
<td>S.PC</td>
<td>msgsndp</td>
<td>Message Send Privileged</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C000120</td>
<td>107</td>
<td>B</td>
<td>mtdcr</td>
<td>Move To Condition Register Fields</td>
<td></td>
</tr>
<tr>
<td>XFX</td>
<td>7C000386</td>
<td>1054</td>
<td>E.DC</td>
<td>mtdcr</td>
<td>Move To Device Control Register</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000346</td>
<td>113</td>
<td>E.DC</td>
<td>mtdcrux</td>
<td>Move To Device Control Register User Mode Indexed</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7C000306</td>
<td>1054</td>
<td>E.DC</td>
<td>mtdcrx</td>
<td>Move To Device Control Register Indexed</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00008C</td>
<td></td>
<td>FP[R]</td>
<td>163</td>
<td>FPSCR</td>
<td>mtfsb0[]</td>
</tr>
<tr>
<td>X</td>
<td>63 FC00004C</td>
<td></td>
<td>FP[R]</td>
<td>163</td>
<td>FPSCR</td>
<td>mtfsb1[]</td>
</tr>
<tr>
<td>XFL</td>
<td>63 FC00058E</td>
<td></td>
<td>FP[R]</td>
<td>162</td>
<td>FPSCR</td>
<td>mtfsl[]</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000124</td>
<td>P</td>
<td>S</td>
<td>884</td>
<td>FPSCR</td>
<td>mtmsr</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000164</td>
<td>P</td>
<td>S</td>
<td>886</td>
<td>FPSCR</td>
<td>mtmsrd</td>
</tr>
<tr>
<td>X</td>
<td>31 7C100120</td>
<td>B</td>
<td></td>
<td>112</td>
<td>FPSCR</td>
<td>mtocrf</td>
</tr>
<tr>
<td>XFX</td>
<td>31 7C00039C</td>
<td>O</td>
<td>E.PM</td>
<td>1257</td>
<td>FPSCR</td>
<td>mtpmr</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000126</td>
<td>S</td>
<td></td>
<td>108</td>
<td></td>
<td>mtsle</td>
</tr>
<tr>
<td>XFX</td>
<td>31 7C0003A6</td>
<td>O</td>
<td>B</td>
<td>104</td>
<td>FPSCR</td>
<td>mtspr</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001A4</td>
<td>P</td>
<td>S</td>
<td>926</td>
<td></td>
<td>mtssr</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001E4</td>
<td>P</td>
<td>S</td>
<td>926</td>
<td></td>
<td>mtssrn</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000644</td>
<td>V</td>
<td></td>
<td>268</td>
<td></td>
<td>mtvscr</td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C000166</td>
<td></td>
<td>VSX</td>
<td>110</td>
<td></td>
<td>mtvsrcd</td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C0001A6</td>
<td></td>
<td>VSX</td>
<td>110</td>
<td></td>
<td>mtvsrwla</td>
</tr>
<tr>
<td>XX2</td>
<td>31 7C0001E6</td>
<td></td>
<td>VSX</td>
<td>111</td>
<td></td>
<td>mtvsrwz</td>
</tr>
<tr>
<td>X</td>
<td>4 10000150</td>
<td>LMA</td>
<td></td>
<td>680</td>
<td></td>
<td>mulchwl[]</td>
</tr>
<tr>
<td>X</td>
<td>4 10000110</td>
<td>LMA</td>
<td></td>
<td>680</td>
<td></td>
<td>mulchwu[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00092</td>
<td>SR</td>
<td>75</td>
<td>64</td>
<td></td>
<td>mulhd[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00012</td>
<td>SR</td>
<td>64</td>
<td>64</td>
<td></td>
<td>mulhdu[]</td>
</tr>
<tr>
<td>X</td>
<td>4 10000050</td>
<td>LMA</td>
<td></td>
<td>681</td>
<td></td>
<td>mulhhw[]</td>
</tr>
<tr>
<td>X</td>
<td>4 10000010</td>
<td>LMA</td>
<td></td>
<td>681</td>
<td></td>
<td>mulhhwu[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00096</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td></td>
<td>mulhw[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C00016</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td></td>
<td>mulhwu[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D2</td>
<td>SR</td>
<td>64</td>
<td>64</td>
<td></td>
<td>mulhd[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0005D2</td>
<td>SR</td>
<td>64</td>
<td>64</td>
<td></td>
<td>mulldo[]</td>
</tr>
<tr>
<td>X</td>
<td>4 10000350</td>
<td>LMA</td>
<td></td>
<td>681</td>
<td></td>
<td>mulhhw[]</td>
</tr>
<tr>
<td>X</td>
<td>4 10000310</td>
<td>LMA</td>
<td></td>
<td>681</td>
<td></td>
<td>mulhhwu[]</td>
</tr>
<tr>
<td>D</td>
<td>7 1C000000</td>
<td></td>
<td>71</td>
<td>B</td>
<td></td>
<td>mull[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0001D6</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td></td>
<td>mulw[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0005D6</td>
<td>SR</td>
<td>71</td>
<td>B</td>
<td></td>
<td>mulw[]</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003B8</td>
<td>SR</td>
<td>84</td>
<td>B</td>
<td></td>
<td>nand[]</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000364</td>
<td>H</td>
<td>867</td>
<td>S</td>
<td></td>
<td>nap</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0000D0</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td></td>
<td>neg[]</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C0004D0</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td></td>
<td>nego[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 1000015C</td>
<td>LMA</td>
<td></td>
<td>682</td>
<td></td>
<td>nmacchwl[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 1000015C</td>
<td>LMA</td>
<td></td>
<td>682</td>
<td></td>
<td>nmacchwo[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 100001DC</td>
<td>LMA</td>
<td></td>
<td>682</td>
<td></td>
<td>nmacchws[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 100001DC</td>
<td>LMA</td>
<td></td>
<td>682</td>
<td></td>
<td>nmacchwo[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 1000005C</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td></td>
<td>nmachhw[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 1000005C</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td></td>
<td>nmachhwo[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 100000DC</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td></td>
<td>nmachhws[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 100000DC</td>
<td>LMA</td>
<td></td>
<td>683</td>
<td></td>
<td>nmachhwo[]</td>
</tr>
<tr>
<td>XO</td>
<td>4 1000035C</td>
<td>LMA</td>
<td></td>
<td>684</td>
<td></td>
<td>nmachhw[]</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>------------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>XO</td>
<td>4 1000035C</td>
<td>684</td>
<td>LMA</td>
<td>nmaclhwo[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Modulo Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 100003DC</td>
<td>684</td>
<td>LMA</td>
<td>nmaclhws[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Saturate Signed</td>
<td></td>
</tr>
<tr>
<td>XO</td>
<td>4 100003DC</td>
<td>684</td>
<td>LMA</td>
<td>nmaclhwso[.]</td>
<td>Negative Multiply Accumulate Low Halfword to Word Saturate Signed &amp; record OV</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000F8</td>
<td>85</td>
<td>B</td>
<td>nor[.]</td>
<td>NOR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000378</td>
<td>84</td>
<td>B</td>
<td>or[.]</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000338</td>
<td>85</td>
<td>B</td>
<td>orc[.]</td>
<td>OR with Complement</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>24 60000000</td>
<td>82</td>
<td>B</td>
<td>ori</td>
<td>OR Immediate</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>25 64000000</td>
<td>83</td>
<td>B</td>
<td>oris</td>
<td>OR Immediate Shifted</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0000F4</td>
<td>87</td>
<td>B</td>
<td>popcntb</td>
<td>Population Count Byte-wise</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003F4</td>
<td>89</td>
<td>64</td>
<td>popcntd</td>
<td>Population Count Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0002F4</td>
<td>87</td>
<td>B</td>
<td>popcntw</td>
<td>Population Count Words</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000174</td>
<td>88</td>
<td>64</td>
<td>prtyd</td>
<td>Parity Doubleword</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000134</td>
<td>88</td>
<td>B</td>
<td>prtyw</td>
<td>Parity Word</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000066</td>
<td>P 1041</td>
<td>E</td>
<td>rfcic</td>
<td>Return From Critical Interrupt</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C00004E</td>
<td>P 1042</td>
<td>E,ED</td>
<td>rfdi</td>
<td>Return From Debug Interrupt</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000124</td>
<td>820</td>
<td>S</td>
<td>rfebb</td>
<td>Return from Event Based Branch</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C0000CC</td>
<td>P 1043</td>
<td>E,HV</td>
<td>rfgi</td>
<td>Return From Guest Interrupt</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000064</td>
<td>P 1041</td>
<td>E</td>
<td>rfi</td>
<td>Return From Interrupt</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C000024</td>
<td>P 864</td>
<td>S</td>
<td>rfid</td>
<td>Return from Interrupt Doubleword</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C00004C</td>
<td>P 1042</td>
<td>E</td>
<td>rfmcic</td>
<td>Return From Machine Check Interrupt</td>
<td></td>
</tr>
<tr>
<td>MDS</td>
<td>30 78000010</td>
<td>SR 95</td>
<td>64</td>
<td>rlclcl[.]</td>
<td>Rotate Left Doubleword then Clear Left</td>
<td></td>
</tr>
<tr>
<td>MDS</td>
<td>30 78000012</td>
<td>SR 96</td>
<td>64</td>
<td>rlclrcl[.]</td>
<td>Rotate Left Doubleword then Clear Right</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>30 78000008</td>
<td>SR 95</td>
<td>64</td>
<td>rlclcl[.]</td>
<td>Rotate Left Doubleword Immediate then Clear Left</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>30 78000000</td>
<td>SR 94</td>
<td>64</td>
<td>rlclcl[.]</td>
<td>Rotate Left Doubleword Immediate then Clear Right</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>30 78000004</td>
<td>SR 94</td>
<td>64</td>
<td>rlclcl[.]</td>
<td>Rotate Left Doubleword Immediate then Clear Right</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>30 7800000C</td>
<td>SR 96</td>
<td>64</td>
<td>rlclcl[.]</td>
<td>Rotate Left Doubleword Immediate then Mask Insert</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>20 50000000</td>
<td>SR 93</td>
<td>B</td>
<td>rlfwim[.]</td>
<td>Rotate Left Word Immediate then Mask Insert</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>21 54000000</td>
<td>SR 91</td>
<td>B</td>
<td>rlfwim[.]</td>
<td>Rotate Left Word Immediate then AND with Mask</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>23 5C000000</td>
<td>SR 92</td>
<td>B</td>
<td>rlfwim[.]</td>
<td>Rotate Left Word then AND with Mask</td>
<td></td>
</tr>
<tr>
<td>XL</td>
<td>19 4C0003E4</td>
<td>H 868</td>
<td>S</td>
<td>rtwinkle</td>
<td>Rip Van Winkle</td>
<td></td>
</tr>
<tr>
<td>SC</td>
<td>17 44000002</td>
<td>863</td>
<td>B</td>
<td>sc</td>
<td>System Call</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007A7</td>
<td>SR</td>
<td>P 923</td>
<td>sibfe,</td>
<td>SLB Find Entry ESID</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0003E4</td>
<td>P 920</td>
<td>S</td>
<td>sibia</td>
<td>SLB Invalidate All</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000364</td>
<td>P 919</td>
<td>S</td>
<td>sibie</td>
<td>SLB Invalidate Entry</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000726</td>
<td>P 923</td>
<td>S</td>
<td>sibmfee</td>
<td>SLB Move From Entry ESID</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006A6</td>
<td>P 922</td>
<td>S</td>
<td>sibmfev</td>
<td>SLB Move From Entry VSID</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000324</td>
<td>P 921</td>
<td>S</td>
<td>sibmte</td>
<td>SLB Move To Entry</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000366</td>
<td>SR</td>
<td>99</td>
<td>64</td>
<td>slid[.]</td>
<td>Shift Left Doubleword</td>
</tr>
<tr>
<td>XL</td>
<td>19 4C0003A4</td>
<td>H 868</td>
<td>S</td>
<td>sleep</td>
<td>Sleep</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31 7C000030</td>
<td>SR</td>
<td>97</td>
<td>B</td>
<td>slw[.]</td>
<td>Shift Left Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000634</td>
<td>SR</td>
<td>100</td>
<td>64</td>
<td>sradi[.]</td>
<td>Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>XS</td>
<td>31 7C000674</td>
<td>SR</td>
<td>100</td>
<td>64</td>
<td>sradi[.]</td>
<td>Shift Right Algebraic Doubleword Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000630</td>
<td>SR</td>
<td>98</td>
<td>B</td>
<td>sraw[.]</td>
<td>Shift Right Algebraic Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000670</td>
<td>SR</td>
<td>98</td>
<td>B</td>
<td>sraw[.]</td>
<td>Shift Right Algebraic Word Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000436</td>
<td>SR</td>
<td>99</td>
<td>64</td>
<td>srd[.]</td>
<td>Shift Right Doubleword</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000430</td>
<td>SR</td>
<td>97</td>
<td>B</td>
<td>srw[.]</td>
<td>Shift Right Word</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege Mode Dep.</td>
<td>Page</td>
<td>Category</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------</td>
<td>---------------------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>D 38</td>
<td>98000000</td>
<td>54 B</td>
<td>B</td>
<td>Store Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0007AA</td>
<td>H 877 S</td>
<td>S</td>
<td>Store Byte Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00056D</td>
<td>779 B</td>
<td>B</td>
<td>Store Byte Conditional Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C000506</td>
<td>823 DS</td>
<td>DS</td>
<td>Store Byte with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0001BE</td>
<td>P 1061 E.PD</td>
<td>E.PD</td>
<td>Store Byte by External PID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 39</td>
<td>9C000000</td>
<td>54 B</td>
<td>B</td>
<td>Store Byte with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0001EE</td>
<td>54 B</td>
<td>B</td>
<td>Store Byte with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0001AE</td>
<td>54 B</td>
<td>B</td>
<td>Store Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS 62</td>
<td>F8000000</td>
<td>57 64 std</td>
<td>std</td>
<td>Store Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C000528</td>
<td>60 64 stdbrx</td>
<td>Store Doubleword Byte-Reverse Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0007EA</td>
<td>H 877 S</td>
<td>S</td>
<td>Store Doubleword Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0001AD</td>
<td>782 64 stdcx.</td>
<td>Store Doubleword Conditional Indexed &amp; record CR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0005C6</td>
<td>823 DS</td>
<td>DS</td>
<td>Store Doubleword with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00013A</td>
<td>P 1062 E.PD;64 stdepx</td>
<td>Store Doubleword by External PID Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS 62</td>
<td>F8000001</td>
<td>57 64 stdu</td>
<td>stdu</td>
<td>Store Doubleword with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00016A</td>
<td>57 64 stdux</td>
<td>Store Doubleword with Update Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00012A</td>
<td>57 64 stdx</td>
<td>stdx</td>
<td>Store Doubleword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 54</td>
<td>D8000000</td>
<td>139 FP</td>
<td>FP</td>
<td>Store Floating-Point Double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C000746</td>
<td>823 DS</td>
<td>DS</td>
<td>Store Floating Doubleword with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 54</td>
<td>D8000001</td>
<td>139 FP</td>
<td>FP</td>
<td>Store Floating-Point Double by External PID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS 61</td>
<td>F4000000</td>
<td>141 FP.out</td>
<td>Store Floating-Point Double Pair</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00072E</td>
<td>141 FP.out</td>
<td>Store Floating-Point Double Pair Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 55</td>
<td>DC000000</td>
<td>139 FP</td>
<td>FP</td>
<td>Store Floating-Point Double with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0005EE</td>
<td>139 FP</td>
<td>FP</td>
<td>Store Floating-Point Double with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0005AE</td>
<td>139 FP</td>
<td>FP</td>
<td>Store Floating-Point Double Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0007AE</td>
<td>140 FP</td>
<td>FP</td>
<td>Store Floating-Point as Integer Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 52</td>
<td>D0000000</td>
<td>138 FP</td>
<td>FP</td>
<td>Store Floating-Point Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 53</td>
<td>D4000000</td>
<td>138 FP</td>
<td>FP</td>
<td>Store Floating-Point Single with Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00056E</td>
<td>138 FP</td>
<td>FP</td>
<td>Store Floating-Point Single with Update Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00052E</td>
<td>138 FP</td>
<td>FP</td>
<td>Store Floating-Point Single Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 44</td>
<td>B0000000</td>
<td>55 B</td>
<td>B</td>
<td>Store Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00072C</td>
<td>59 B</td>
<td>B</td>
<td>Store Halfword Byte-Reverse Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00076A</td>
<td>H 877 S</td>
<td>S</td>
<td>Store Halfword and Zero Caching Inhibited Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0005AD</td>
<td>780 B</td>
<td>B</td>
<td>Store Halfword Conditional Indexed Xform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C000546</td>
<td>823 DS</td>
<td>DS</td>
<td>Store Halword with Decoration Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00033E</td>
<td>P 1061 E.PD</td>
<td>E.PD</td>
<td>Store Halword by External PID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 45</td>
<td>B4000000</td>
<td>55 B</td>
<td>B</td>
<td>Store Halfword with Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00036E</td>
<td>55 B</td>
<td>B</td>
<td>Store Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00032E</td>
<td>55 B</td>
<td>B</td>
<td>Store Halword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 47</td>
<td>BC000000</td>
<td>61 B</td>
<td>B</td>
<td>Store Multiple Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS 62</td>
<td>F8000000</td>
<td>58 LSQ</td>
<td>LSQ</td>
<td>Store Quadword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00016D</td>
<td>785 LSQ</td>
<td>LSQ</td>
<td>Store Quadword Conditional Indexed and record CR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C0005AA</td>
<td>64 MA</td>
<td>MA</td>
<td>Store String Word Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00052A</td>
<td>64 MA</td>
<td>MA</td>
<td>Store String Word Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00010E</td>
<td>184 V</td>
<td>V</td>
<td>Store Vector Element Byte Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00014E</td>
<td>184 V</td>
<td>V</td>
<td>Store Vector Element Halfword Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00064E</td>
<td>P 1071 E.PD</td>
<td>E.PD</td>
<td>Store Vector by External PID Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X 31</td>
<td>7C00060E</td>
<td>P 1071 E.PD</td>
<td>E.PD</td>
<td>Store Vector by External PID Indexed Last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00018E</td>
<td>V</td>
<td>185</td>
<td>S</td>
<td>stvewx</td>
<td>Store Vector Element Word Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0001CE</td>
<td>V</td>
<td>185</td>
<td>S</td>
<td>stvx</td>
<td>Store Vector Indexed</td>
</tr>
<tr>
<td>D</td>
<td>36 90000000</td>
<td>B</td>
<td>56</td>
<td>S</td>
<td>stw</td>
<td>Store Word</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00052C</td>
<td>B</td>
<td>59</td>
<td>S</td>
<td>stwbrx</td>
<td>Store Word Byte-Reverse Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00072A</td>
<td>H</td>
<td>877</td>
<td>S</td>
<td>stwcix</td>
<td>Store Word and Zero Caching Inhibited Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00012D</td>
<td>-</td>
<td>781</td>
<td>B</td>
<td>stwcx</td>
<td>Store Word Conditional Indexed &amp; record CR0</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000586</td>
<td>-</td>
<td>823</td>
<td>DS</td>
<td>stwdx</td>
<td>Store Word with Decoration Indexed</td>
</tr>
<tr>
<td>D</td>
<td>8 20000000</td>
<td>-</td>
<td>809</td>
<td>TM</td>
<td>tabort</td>
<td>Transaction Abort</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000050</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>sub[f]</td>
<td>Subtract From</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000106</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>sub[f][c]</td>
<td>Subtract From Carrying</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000410</td>
<td>SR</td>
<td>68</td>
<td>B</td>
<td>sub[f][c]</td>
<td>Subtract From Carrying &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000110</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfe</td>
<td>Subtract From Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000510</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subfeo</td>
<td>Subtract From Extended &amp; record OV</td>
</tr>
<tr>
<td>D</td>
<td>8 20000000</td>
<td>B</td>
<td>68</td>
<td>B</td>
<td>subfic</td>
<td>Subtract From Immediate Carrying</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000106</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>subme</td>
<td>Subtract From Minus One Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000500</td>
<td>SR</td>
<td>69</td>
<td>B</td>
<td>submeo</td>
<td>Subtract From Minus One Extended &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000450</td>
<td>SR</td>
<td>67</td>
<td>B</td>
<td>subf[z]</td>
<td>Subtract From &amp; record OV</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000190</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>subfzez</td>
<td>Subtract From Zero Extended</td>
</tr>
<tr>
<td>XO</td>
<td>31 7C000590</td>
<td>SR</td>
<td>70</td>
<td>B</td>
<td>subfzeo</td>
<td>Subtract From Zero Extended &amp; record OV</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0004AC</td>
<td>-</td>
<td>786</td>
<td>T</td>
<td>sync</td>
<td>Synchronize</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00071D</td>
<td>TM</td>
<td>808</td>
<td>T</td>
<td>tabort</td>
<td>Transaction Abort</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00065D</td>
<td>TM</td>
<td>809</td>
<td>T</td>
<td>tabortdc</td>
<td>Transaction Abort Doubleword Conditional</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006DD</td>
<td>TM</td>
<td>810</td>
<td>T</td>
<td>tabortdci</td>
<td>Transaction Abort Doubleword Conditional Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00061D</td>
<td>TM</td>
<td>809</td>
<td>T</td>
<td>tabortwc</td>
<td>Transaction Abort Word Conditional</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00069D</td>
<td>TM</td>
<td>809</td>
<td>T</td>
<td>tabortwci</td>
<td>Transaction Abort Word Conditional Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00051D</td>
<td>TM</td>
<td>805</td>
<td>T</td>
<td>tbegin</td>
<td>Transaction Begin</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00059C</td>
<td>TM</td>
<td>811</td>
<td>T</td>
<td>tcheck</td>
<td>Transaction Check</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000088</td>
<td>T</td>
<td>81</td>
<td>64</td>
<td>td</td>
<td>Trap Doubleword</td>
</tr>
<tr>
<td>D</td>
<td>8 08000000</td>
<td>T</td>
<td>81</td>
<td>64</td>
<td>td</td>
<td>Trap Doubleword Immediate</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00055C</td>
<td>T</td>
<td>807</td>
<td>T</td>
<td>tend</td>
<td>Transaction End</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0002E4</td>
<td>H</td>
<td>933</td>
<td>S</td>
<td>tlbia</td>
<td>TLB Invalidate All</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000264</td>
<td>64</td>
<td>928</td>
<td>S</td>
<td>tlbie</td>
<td>TLB Invalidate Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000224</td>
<td>64</td>
<td>931</td>
<td>S</td>
<td>tlibel</td>
<td>TLB Invalidate Entry Local</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000224</td>
<td>P</td>
<td>1134</td>
<td>E</td>
<td>tlbil</td>
<td>TLB Invalidate Local Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000624</td>
<td>P</td>
<td>1132</td>
<td>E</td>
<td>tlbivax</td>
<td>TLB Invalidate Virtual Address Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000764</td>
<td>P</td>
<td>1139</td>
<td>E</td>
<td>tlbre</td>
<td>TLB Read Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0006A5</td>
<td>P</td>
<td>1138</td>
<td>E</td>
<td>tbrsx</td>
<td>TLB Search and Reserve Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C000724</td>
<td>P</td>
<td>1136</td>
<td>E</td>
<td>tblsx</td>
<td>TLB Search Indexed</td>
</tr>
<tr>
<td>X</td>
<td>31 7C00046C</td>
<td>H</td>
<td>934</td>
<td>S</td>
<td>tblsync</td>
<td>TLB Synchronize</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007A4</td>
<td>P</td>
<td>1141</td>
<td>E</td>
<td>tblwe</td>
<td>TLB Write Entry</td>
</tr>
<tr>
<td>X</td>
<td>31 7C0007DD</td>
<td>-</td>
<td>880</td>
<td>TM</td>
<td>trechkpt</td>
<td>Transaction Recheckpoint</td>
</tr>
</tbody>
</table>

Appendix I. Power ISA Instruction Set Sorted by Mnemonic
<table>
<thead>
<tr>
<th>Format</th>
<th>Opcode</th>
<th>Mode Dep.</th>
<th>Privilege</th>
<th>Page</th>
<th>Category</th>
<th>Mnemonic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>31</td>
<td>7C00075D</td>
<td>TM</td>
<td>879</td>
<td>Tc reclaim.</td>
<td>Transaction Reclaim</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C0005DC</td>
<td>TM</td>
<td>810</td>
<td>tsr.</td>
<td>Transaction Suspend or Resume</td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>7C000008</td>
<td>B</td>
<td>80</td>
<td>tw</td>
<td>Trap Word</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>0C000000</td>
<td>B</td>
<td>80</td>
<td>twi</td>
<td>Trap Word Immediate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000140</td>
<td>V</td>
<td>206</td>
<td>vaddcuq</td>
<td>Vector Add &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000180</td>
<td>V</td>
<td>202</td>
<td>vaddcww</td>
<td>Vector Add and Write Carry-Out Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000003D</td>
<td>V</td>
<td>206</td>
<td>vaddcuq</td>
<td>Vector Add Extended &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>VA</td>
<td>4</td>
<td>1000003C</td>
<td>V</td>
<td>206</td>
<td>vaddcuqm</td>
<td>Vector Add Extended Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000000A</td>
<td>V</td>
<td>244</td>
<td>vaddfp</td>
<td>Vector Add Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000300</td>
<td>V</td>
<td>202</td>
<td>vaddsbds</td>
<td>Vector Add Signed Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000340</td>
<td>V</td>
<td>202</td>
<td>vaddshs</td>
<td>Vector Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000380</td>
<td>V</td>
<td>203</td>
<td>vaddsws</td>
<td>Vector Add Signed Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000000</td>
<td>V</td>
<td>203</td>
<td>vaddubm</td>
<td>Vector Add Unsigned Byte Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000200</td>
<td>V</td>
<td>205</td>
<td>vaddubs</td>
<td>Vector Add Unsigned Byte Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000000C</td>
<td>V</td>
<td>203</td>
<td>vaddudm</td>
<td>Vector Add Unsigned Doubleword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000040</td>
<td>V</td>
<td>203</td>
<td>vadduhm</td>
<td>Vector Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000240</td>
<td>V</td>
<td>205</td>
<td>vadduhs</td>
<td>Vector Add Unsigned Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000100</td>
<td>V</td>
<td>206</td>
<td>vadduqm</td>
<td>Vector Add Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000080</td>
<td>V</td>
<td>204</td>
<td>vadduwm</td>
<td>Vector Add Unsigned Word Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000280</td>
<td>V</td>
<td>205</td>
<td>vadduws</td>
<td>Vector Add Unsigned Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000404</td>
<td>V</td>
<td>238</td>
<td>vand</td>
<td>Vector Logical AND</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000444</td>
<td>V</td>
<td>238</td>
<td>vandc</td>
<td>Vector Logical AND with Complement</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000502</td>
<td>V</td>
<td>226</td>
<td>vavgbsb</td>
<td>Vector Average Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000542</td>
<td>V</td>
<td>226</td>
<td>vavgsh</td>
<td>Vector Average Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000582</td>
<td>V</td>
<td>226</td>
<td>vavgsw</td>
<td>Vector Average Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000402</td>
<td>V</td>
<td>227</td>
<td>vavgubs</td>
<td>Vector Average Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000442</td>
<td>V</td>
<td>227</td>
<td>vavguh</td>
<td>Vector Average Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000482</td>
<td>V</td>
<td>227</td>
<td>vavguw</td>
<td>Vector Average Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000054C</td>
<td>V</td>
<td>265</td>
<td>vbpermq</td>
<td>Vector Bit Permute Quadword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000054C</td>
<td>V</td>
<td>265</td>
<td>vbpermq</td>
<td>Vector Bit Permute Quadword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000034A</td>
<td>V</td>
<td>248</td>
<td>vcfxs</td>
<td>Vector Convert From Signed Fixed-Point Word To Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>1000030A</td>
<td>V</td>
<td>248</td>
<td>vcfux</td>
<td>Vector Convert From Unsigned Fixed-Point Word To Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000508</td>
<td>V.Crypto</td>
<td>256</td>
<td>vcipher</td>
<td>Vector AES Cipher</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000509</td>
<td>V.Crypto</td>
<td>256</td>
<td>vcipherlast</td>
<td>Vector AES Cipher Last</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000702</td>
<td>V</td>
<td>263</td>
<td>vclzb</td>
<td>Vector Count Leading Zeros Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>100007C2</td>
<td>V</td>
<td>263</td>
<td>vclzd</td>
<td>Vector Count Leading Zeros Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000742</td>
<td>V</td>
<td>263</td>
<td>vclzh</td>
<td>Vector Count Leading Zeros Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td>10000782</td>
<td>V</td>
<td>263</td>
<td>vclzw</td>
<td>Vector Count Leading Zeros Word</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100003C6</td>
<td>V</td>
<td>251</td>
<td>vcmpbf[p]</td>
<td>Vector Compare Bounds Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100000C6</td>
<td>V</td>
<td>252</td>
<td>vcmpqfp[p]</td>
<td>Vector Compare Equal To Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000006</td>
<td>V</td>
<td>232</td>
<td>vcmpqub[p]</td>
<td>Vector Compare Equal To Unsigned Byte</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100000C7</td>
<td>V</td>
<td>233</td>
<td>vcmpqaud[p]</td>
<td>Vector Compare Equal To Unsigned Doubleword</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000046</td>
<td>V</td>
<td>233</td>
<td>vcmpqueq[p]</td>
<td>Vector Compare Equal To Unsigned Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000086</td>
<td>V</td>
<td>233</td>
<td>vcmpquew[p]</td>
<td>Vector Compare Equal To Unsigned Word</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100001C6</td>
<td>V</td>
<td>252</td>
<td>vcmpgefp[p]</td>
<td>Vector Compare Greater Than or Equal To Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>100002C6</td>
<td>V</td>
<td>253</td>
<td>vcmpgtflp[p]</td>
<td>Vector Compare Greater Than Single-Precision</td>
</tr>
<tr>
<td>VC</td>
<td>4</td>
<td>10000306</td>
<td>V</td>
<td>234</td>
<td>vcmpgtsb[p]</td>
<td>Vector Compare Greater Than Signed Byte</td>
</tr>
<tr>
<td>Format</td>
<td>Primary Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>VC</td>
<td>4 100003C7</td>
<td>V</td>
<td>234</td>
<td>V</td>
<td>vcmpgtsd[]</td>
<td>Vector Compare Greater Than Signed Doubleword</td>
</tr>
<tr>
<td>VC</td>
<td>4 10000346</td>
<td>V</td>
<td>234</td>
<td>V</td>
<td>vcmpgtsh[]</td>
<td>Vector Compare Greater Than Signed Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>4 10000386</td>
<td>V</td>
<td>235</td>
<td>V</td>
<td>vcmpgtsw[]</td>
<td>Vector Compare Greater Than Signed Word</td>
</tr>
<tr>
<td>VC</td>
<td>4 10000206</td>
<td>V</td>
<td>236</td>
<td>V</td>
<td>vcmpgtub[]</td>
<td>Vector Compare Greater Than Unsigned Byte</td>
</tr>
<tr>
<td>VC</td>
<td>4 100002C7</td>
<td>V</td>
<td>236</td>
<td>V</td>
<td>vcmpgtud[]</td>
<td>Vector Compare Greater Than Unsigned Doubleword</td>
</tr>
<tr>
<td>VC</td>
<td>4 10000246</td>
<td>V</td>
<td>236</td>
<td>V</td>
<td>vcmpgtuh[]</td>
<td>Vector Compare Greater Than Unsigned Halfword</td>
</tr>
<tr>
<td>VC</td>
<td>4 10000286</td>
<td>V</td>
<td>237</td>
<td>V</td>
<td>vcmpgtuw[]</td>
<td>Vector Compare Greater Than Unsigned Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 100003CA</td>
<td>V</td>
<td>247</td>
<td>V</td>
<td>vctxs</td>
<td>Vector Convert From Single-Precision To Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000038A</td>
<td>V</td>
<td>247</td>
<td>V</td>
<td>vctuxs</td>
<td>Vector Convert From Single-Precision To Unsigned Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000684</td>
<td>V</td>
<td>238</td>
<td>V</td>
<td>veqv</td>
<td>Vector Equivalence</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000018A</td>
<td>V</td>
<td>254</td>
<td>V</td>
<td>vexptefp</td>
<td>Vector 2 Raised to the Exponent Estimate Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 100005OC</td>
<td>V</td>
<td>262</td>
<td>V</td>
<td>vgbbd</td>
<td>Vector Gather Bits by Byte by Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 100001CA</td>
<td>V</td>
<td>254</td>
<td>V</td>
<td>vlogfp</td>
<td>Vector Log Base 2 Estimate Single-Precision</td>
</tr>
<tr>
<td>VA</td>
<td>4 1000002E</td>
<td>V</td>
<td>245</td>
<td>V</td>
<td>vmaddfp</td>
<td>Vector Multiply-Add Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000040A</td>
<td>V</td>
<td>246</td>
<td>V</td>
<td>vmaxfp</td>
<td>Vector Maximum Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000102</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmAXB</td>
<td>Vector Maximum Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 100001C2</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxsd</td>
<td>Vector Maximum Signed Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000142</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxsh</td>
<td>Vector Maximum Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000182</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxsw</td>
<td>Vector Maximum Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000002</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxub</td>
<td>Vector Maximum Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 100000C2</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxud</td>
<td>Vector Maximum Unsigned Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000042</td>
<td>V</td>
<td>228</td>
<td>V</td>
<td>vmaxuh</td>
<td>Vector Maximum Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000082</td>
<td>V</td>
<td>229</td>
<td>V</td>
<td>vmaxuw</td>
<td>Vector Maximum Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000020</td>
<td>V</td>
<td>218</td>
<td>V</td>
<td>vmhaddshs</td>
<td>Vector Multiply-High-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000021</td>
<td>V</td>
<td>218</td>
<td>V</td>
<td>vmhraddshs</td>
<td>Vector Multiply-High-Round-Add Signed Halfword Saturate</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000044A</td>
<td>V</td>
<td>246</td>
<td>V</td>
<td>vmninfp</td>
<td>Vector Minimum Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000302</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vmnisb</td>
<td>Vector Minimum Signed Byte</td>
</tr>
<tr>
<td>X</td>
<td>4 100003C2</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vmnsd</td>
<td>Vector Minimum Signed Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000342</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vmnsh</td>
<td>Vector Minimum Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000382</td>
<td>V</td>
<td>231</td>
<td>V</td>
<td>vmnsw</td>
<td>Vector Minimum Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000202</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vmnub</td>
<td>Vector Minimum Unsigned Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 100002C2</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vmnud</td>
<td>Vector Minimum Unsigned Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000242</td>
<td>V</td>
<td>230</td>
<td>V</td>
<td>vmnuh</td>
<td>Vector Minimum Unsigned Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 10000282</td>
<td>V</td>
<td>231</td>
<td>V</td>
<td>vnumuw</td>
<td>Vector Minimum Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000022</td>
<td>V</td>
<td>219</td>
<td>V</td>
<td>vmladduhm</td>
<td>Vector Multiply-Low-Add Unsigned Halfword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000078C</td>
<td>VSX</td>
<td>196</td>
<td>VSX</td>
<td>vmrgew</td>
<td>Vector Merge Even Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000000C</td>
<td>V</td>
<td>194</td>
<td>V</td>
<td>vmrghb</td>
<td>Vector Merge High Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000004C</td>
<td>V</td>
<td>194</td>
<td>V</td>
<td>vmrghh</td>
<td>Vector Merge High Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000008C</td>
<td>V</td>
<td>195</td>
<td>V</td>
<td>vmrghw</td>
<td>Vector Merge High Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000010C</td>
<td>V</td>
<td>194</td>
<td>V</td>
<td>vmrglb</td>
<td>Vector Merge Low Byte</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000014C</td>
<td>V</td>
<td>194</td>
<td>V</td>
<td>vmrglh</td>
<td>Vector Merge Low Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000018C</td>
<td>V</td>
<td>195</td>
<td>V</td>
<td>vmrglw</td>
<td>Vector Merge Low Word</td>
</tr>
<tr>
<td>VX</td>
<td>4 1000068C</td>
<td>VSX</td>
<td>196</td>
<td>VSX</td>
<td>vmrgow</td>
<td>Vector Merge Odd Word</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000025</td>
<td>V</td>
<td>220</td>
<td>V</td>
<td>vmsummbm</td>
<td>Vector Multiply-Sum Mixed Byte Modulo</td>
</tr>
<tr>
<td>VA</td>
<td>4 10000028</td>
<td>V</td>
<td>220</td>
<td>V</td>
<td>vmsumshm</td>
<td>Vector Multiply-Sum Signed Halfword Modulo</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>VA</td>
<td>10000029</td>
<td>V</td>
<td>221</td>
<td></td>
<td></td>
<td>vmsumshs</td>
</tr>
<tr>
<td>VA</td>
<td>10000024</td>
<td>V</td>
<td>219</td>
<td></td>
<td></td>
<td>vmsumubm</td>
</tr>
<tr>
<td>VA</td>
<td>10000026</td>
<td>V</td>
<td>221</td>
<td></td>
<td></td>
<td>vmsumuhm</td>
</tr>
<tr>
<td>VA</td>
<td>10000027</td>
<td>V</td>
<td>222</td>
<td></td>
<td></td>
<td>vmsumuh</td>
</tr>
<tr>
<td>VX</td>
<td>10000308</td>
<td>V</td>
<td>214</td>
<td></td>
<td></td>
<td>vmulesb</td>
</tr>
<tr>
<td>VX</td>
<td>10000348</td>
<td>V</td>
<td>215</td>
<td></td>
<td></td>
<td>vmulesh</td>
</tr>
<tr>
<td>VX</td>
<td>10000388</td>
<td>V</td>
<td>216</td>
<td></td>
<td></td>
<td>vmulesw</td>
</tr>
<tr>
<td>VX</td>
<td>10000208</td>
<td>V</td>
<td>214</td>
<td></td>
<td></td>
<td>vmuleub</td>
</tr>
<tr>
<td>VX</td>
<td>10000248</td>
<td>V</td>
<td>215</td>
<td></td>
<td></td>
<td>vmuleuh</td>
</tr>
<tr>
<td>VX</td>
<td>10000288</td>
<td>V</td>
<td>216</td>
<td></td>
<td></td>
<td>vmuleuw</td>
</tr>
<tr>
<td>VX</td>
<td>10000108</td>
<td>V</td>
<td>214</td>
<td></td>
<td></td>
<td>vmulosb</td>
</tr>
<tr>
<td>VX</td>
<td>10000148</td>
<td>V</td>
<td>215</td>
<td></td>
<td></td>
<td>vmulosh</td>
</tr>
<tr>
<td>VX</td>
<td>10000188</td>
<td>V</td>
<td>216</td>
<td></td>
<td></td>
<td>vmulosw</td>
</tr>
<tr>
<td>VX</td>
<td>10000008</td>
<td>V</td>
<td>214</td>
<td></td>
<td></td>
<td>vmuloub</td>
</tr>
<tr>
<td>VX</td>
<td>10000048</td>
<td>V</td>
<td>215</td>
<td></td>
<td></td>
<td>vmulouh</td>
</tr>
<tr>
<td>VX</td>
<td>10000088</td>
<td>V</td>
<td>216</td>
<td></td>
<td></td>
<td>vmulouw</td>
</tr>
<tr>
<td>VX</td>
<td>10000089</td>
<td>V</td>
<td>217</td>
<td></td>
<td></td>
<td>vmuluw</td>
</tr>
<tr>
<td>VX</td>
<td>10000584</td>
<td>V</td>
<td>238</td>
<td></td>
<td></td>
<td>vnand</td>
</tr>
<tr>
<td>VX</td>
<td>10000548</td>
<td>V</td>
<td>257</td>
<td></td>
<td></td>
<td>vnmsubfp</td>
</tr>
<tr>
<td>VX</td>
<td>10000549</td>
<td>V</td>
<td>257</td>
<td></td>
<td></td>
<td>vnmsubfp</td>
</tr>
<tr>
<td>VA</td>
<td>1000002F</td>
<td>V</td>
<td>245</td>
<td></td>
<td></td>
<td>vnmsubfp</td>
</tr>
<tr>
<td>VX</td>
<td>10000504</td>
<td>V</td>
<td>239</td>
<td></td>
<td></td>
<td>vnor</td>
</tr>
<tr>
<td>VX</td>
<td>10000484</td>
<td>V</td>
<td>239</td>
<td></td>
<td></td>
<td>vor</td>
</tr>
<tr>
<td>VX</td>
<td>10000544</td>
<td>V</td>
<td>239</td>
<td></td>
<td></td>
<td>vorc</td>
</tr>
<tr>
<td>VA</td>
<td>1000002B</td>
<td>V</td>
<td>198</td>
<td></td>
<td></td>
<td>vperm</td>
</tr>
<tr>
<td>VX</td>
<td>1000002D</td>
<td>V</td>
<td>261</td>
<td></td>
<td></td>
<td>vperm</td>
</tr>
<tr>
<td>VX</td>
<td>1000030E</td>
<td>V</td>
<td>187</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>100005CE</td>
<td>V</td>
<td>187</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000054E</td>
<td>V</td>
<td>188</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>100001BE</td>
<td>V</td>
<td>188</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000010E</td>
<td>V</td>
<td>189</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>100001CE</td>
<td>V</td>
<td>189</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000014E</td>
<td>V</td>
<td>190</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000044E</td>
<td>V</td>
<td>190</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>100004CE</td>
<td>V</td>
<td>190</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000000E</td>
<td>V</td>
<td>190</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000008E</td>
<td>V</td>
<td>191</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000004E</td>
<td>V</td>
<td>191</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>100000CE</td>
<td>V</td>
<td>191</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000040E</td>
<td>V</td>
<td>259</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>100004CE</td>
<td>V</td>
<td>259</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000048E</td>
<td>V</td>
<td>260</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>1000048E</td>
<td>V</td>
<td>260</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>VX</td>
<td>10000703</td>
<td>V</td>
<td>264</td>
<td></td>
<td></td>
<td>vpkmx</td>
</tr>
<tr>
<td>Format</td>
<td>Primary Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>VX</td>
<td>100007C3</td>
<td>V</td>
<td></td>
<td>264</td>
<td>V</td>
<td>vpopcntd Vector Population Count Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>10000743</td>
<td>V</td>
<td></td>
<td>264</td>
<td>V</td>
<td>vpopcnth Vector Population Count Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000783</td>
<td>V</td>
<td></td>
<td>264</td>
<td>V</td>
<td>vpopcntw Vector Population Count Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000010A</td>
<td>V</td>
<td></td>
<td>255</td>
<td>V</td>
<td>vrefp Vector Reciprocal Estimate Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>100002CA</td>
<td>V</td>
<td></td>
<td>250</td>
<td>V</td>
<td>vrfin Vector Round to Single-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>VX</td>
<td>1000020A</td>
<td>V</td>
<td></td>
<td>249</td>
<td>V</td>
<td>vrfin Vector Round to Single-Precision Integer Nearest</td>
</tr>
<tr>
<td>VX</td>
<td>1000028A</td>
<td>V</td>
<td></td>
<td>249</td>
<td>V</td>
<td>vrfip Vector Round to Single-Precision Integer toward +Infinity</td>
</tr>
<tr>
<td>VX</td>
<td>1000024A</td>
<td>V</td>
<td></td>
<td>249</td>
<td>V</td>
<td>vrfiz Vector Round to Single-Precision Integer toward Zero</td>
</tr>
<tr>
<td>VX</td>
<td>10000004</td>
<td>V</td>
<td></td>
<td>240</td>
<td>V</td>
<td>vrlb Vector Rotate Left Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000000C4</td>
<td>V</td>
<td></td>
<td>240</td>
<td>V</td>
<td>vrlb Vector Rotate Left Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>100000044</td>
<td>V</td>
<td></td>
<td>240</td>
<td>V</td>
<td>vrlh Vector Rotate Left Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000084</td>
<td>V</td>
<td></td>
<td>240</td>
<td>V</td>
<td>vrlw Vector Rotate Left Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000014A</td>
<td>V</td>
<td></td>
<td>255</td>
<td>V</td>
<td>vrsrtefp Vector Reciprocal Square Root Estimate Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>100005C8</td>
<td>V</td>
<td></td>
<td>257</td>
<td>V</td>
<td>vrsbox Vector AES S-Box</td>
</tr>
<tr>
<td>VA</td>
<td>1000002A</td>
<td>V</td>
<td></td>
<td>199</td>
<td>V</td>
<td>vsel Vector Select</td>
</tr>
<tr>
<td>VX</td>
<td>1000006C2</td>
<td>V</td>
<td></td>
<td>258</td>
<td>V</td>
<td>vsel Vector Shift Left Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000006C2</td>
<td>V</td>
<td></td>
<td>258</td>
<td>V</td>
<td>vshasigm Vector SHA-512 Sigma Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>1000006C2</td>
<td>V</td>
<td></td>
<td>258</td>
<td>V</td>
<td>vshasigmaw Vector SHA-256 Sigma Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000001C4</td>
<td>V</td>
<td></td>
<td>200</td>
<td>V</td>
<td>vsl Vector Shift Left</td>
</tr>
<tr>
<td>VX</td>
<td>10000104</td>
<td>V</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vslb Vector Shift Left Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000005C4</td>
<td>V</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vsl Vector Shift Left Doubleword</td>
</tr>
<tr>
<td>VA</td>
<td>1000002C</td>
<td>V</td>
<td></td>
<td>200</td>
<td>V</td>
<td>vslidl Vector Shift Left Double by Octet Immediate</td>
</tr>
<tr>
<td>VX</td>
<td>10000144</td>
<td>V</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vslh Vector Shift Left Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>1000004C</td>
<td>V</td>
<td></td>
<td>200</td>
<td>V</td>
<td>vsl Vector Shift Left by Octet</td>
</tr>
<tr>
<td>VX</td>
<td>10000184</td>
<td>V</td>
<td></td>
<td>241</td>
<td>V</td>
<td>vslw Vector Shift Left Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000020C</td>
<td>V</td>
<td></td>
<td>197</td>
<td>V</td>
<td>vspltb Vector Splat Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000020C</td>
<td>V</td>
<td></td>
<td>197</td>
<td>V</td>
<td>vsplth Vector Splat Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>1000030C</td>
<td>V</td>
<td></td>
<td>198</td>
<td>V</td>
<td>vspltsb Vector Splat Immediate Signed Byte</td>
</tr>
<tr>
<td>VX</td>
<td>1000034C</td>
<td>V</td>
<td></td>
<td>198</td>
<td>V</td>
<td>vspltsb Vector Splat Immediate Signed Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>1000038C</td>
<td>V</td>
<td></td>
<td>198</td>
<td>V</td>
<td>vspltsw Vector Splat Immediate Signed Word</td>
</tr>
<tr>
<td>VX</td>
<td>1000028C</td>
<td>V</td>
<td></td>
<td>197</td>
<td>V</td>
<td>vspltw Vector Splat Word</td>
</tr>
<tr>
<td>VX</td>
<td>100002C4</td>
<td>V</td>
<td></td>
<td>201</td>
<td>V</td>
<td>vsr Vector Shift Right</td>
</tr>
<tr>
<td>VX</td>
<td>10000304</td>
<td>V</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsrab Vector Shift Right Algebraic Byte</td>
</tr>
<tr>
<td>VX</td>
<td>100003C4</td>
<td>V</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsrad Vector Shift Right Algebraic Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>10000344</td>
<td>V</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsrah Vector Shift Right Algebraic Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>10000384</td>
<td>V</td>
<td></td>
<td>243</td>
<td>V</td>
<td>vsraw Vector Shift Right Algebraic Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000204</td>
<td>V</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrb Vector Shift Right Byte</td>
</tr>
<tr>
<td>VX</td>
<td>100006C4</td>
<td>V</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrd Vector Shift Right Doubleword</td>
</tr>
<tr>
<td>VX</td>
<td>10000244</td>
<td>V</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrh Vector Shift Right Halfword</td>
</tr>
<tr>
<td>VX</td>
<td>1000044C</td>
<td>V</td>
<td></td>
<td>201</td>
<td>V</td>
<td>vsro Vector Shift Right by Octet</td>
</tr>
<tr>
<td>VX</td>
<td>10000284</td>
<td>V</td>
<td></td>
<td>242</td>
<td>V</td>
<td>vsrw Vector Shift Right Word</td>
</tr>
<tr>
<td>VX</td>
<td>10000540</td>
<td>V</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsubcuq Vector Subtract &amp; write Carry Unsigned Quadword</td>
</tr>
<tr>
<td>VX</td>
<td>10000580</td>
<td>V</td>
<td></td>
<td>208</td>
<td>V</td>
<td>vsubcuw Vector Subtract and Write Carry-Out Unsigned Word</td>
</tr>
<tr>
<td>VA</td>
<td>1000003F</td>
<td>V</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsubecuq Vector Subtract Extended &amp; write CarryUnsigned Quadword</td>
</tr>
<tr>
<td>VA</td>
<td>1000003E</td>
<td>V</td>
<td></td>
<td>212</td>
<td>V</td>
<td>vsubeuqm Vector Subtract Extended Unsigned Quadword Modulo</td>
</tr>
<tr>
<td>VX</td>
<td>1000004A</td>
<td>V</td>
<td></td>
<td>244</td>
<td>V</td>
<td>vsubfp Vector Subtract Single-Precision</td>
</tr>
<tr>
<td>VX</td>
<td>10000700</td>
<td>V</td>
<td></td>
<td>208</td>
<td>V</td>
<td>vsubsbs Vector Subtract Signed Byte Saturate</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Page</td>
<td>Mnemonic</td>
<td>Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubshs</td>
<td>Vector Subtract Signed Halfword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubsws</td>
<td>Vector Subtract Signed Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsububm</td>
<td>Vector Subtract Unsigned Byte Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsububs</td>
<td>Vector Subtract Unsigned Byte Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubudm</td>
<td>Vector Subtract Unsigned Doubleword Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubuhm</td>
<td>Vector Subtract Unsigned Halfword Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubuhws</td>
<td>Vector SubtractUnsigned Halfword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubuwm</td>
<td>Vector Subtract Unsigned Word Modulo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsubuws</td>
<td>Vector Subtract Unsigned Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsum2sws</td>
<td>Vector Sum across Half Signed Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsum4sbs</td>
<td>Vector Sum across Quarter Signed Byte Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsum4shs</td>
<td>Vector Sum across Quarter Signed Halfword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsum4ubs</td>
<td>Vector Sum across Quarter Unsigned Byte Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vsum4ws</td>
<td>Vector Sum across Signed Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupkhpix</td>
<td>Vector Unpack High Pixel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupkhsb</td>
<td>Vector Unpack High Signed Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupkshs</td>
<td>Vector Unpack High Signed Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupkhsx</td>
<td>Vector Unpack High Signed Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupklpx</td>
<td>Vector Unpack Low Pixel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupklxs</td>
<td>Vector Unpack Low Signed Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupklsh</td>
<td>Vector Unpack Low Signed Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vupklsw</td>
<td>Vector Unpack Low Signed Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VX</td>
<td>4</td>
<td></td>
<td>vxor</td>
<td>Vector Logical XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td></td>
<td>wait</td>
<td>Wait for Interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>P</td>
<td>wrtiee</td>
<td>Write External Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td>P</td>
<td>wrtieei</td>
<td>Write External Enable Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>26</td>
<td></td>
<td>xnop</td>
<td>Executed No Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>31</td>
<td></td>
<td>xor[.]</td>
<td>XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>26</td>
<td></td>
<td>xori</td>
<td>XOR Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>27</td>
<td></td>
<td>xoris</td>
<td>XOR Immediate Shifted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xsabsdp</td>
<td>VSX Scalar Absolute Value Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td></td>
<td>xsadddp</td>
<td>VSX Scalar Add Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td></td>
<td>xsaddsp</td>
<td>VSX Scalar Add Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td></td>
<td>xscmpdp</td>
<td>VSX Scalar Compare Ordered Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td></td>
<td>xscompupdp</td>
<td>VSX Scalar Compare Unordered Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td></td>
<td>xs CPSGNDP</td>
<td>VSX Scalar Copy Sign Double-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvdpss</td>
<td>VSX Scalar Convert Double-Precision to Single-Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvdpspn</td>
<td>VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvdpssxsds</td>
<td>VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvdpssxws</td>
<td>VSX Scalar Convert Double-Precision to Signed Fixed-Point Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvdpuxds</td>
<td>VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvdpuxws</td>
<td>VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Word Saturate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td></td>
<td>xscvspdp</td>
<td>VSX Scalar Convert Single-Precision to Double-Precision (p=1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Opcode Primary Base (hex)</td>
<td>Mode Dep.1</td>
<td>Privilege1</td>
<td>Page</td>
<td>Category1</td>
<td>Mnemonic Instruction</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------</td>
<td>------------</td>
<td>-----------</td>
<td>------</td>
<td>-----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000052C</td>
<td></td>
<td></td>
<td>422</td>
<td>VSX</td>
<td>xscvspdnp Scalar Convert Single-Precision to Double-Precision format Non-signalling</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00005E0</td>
<td></td>
<td></td>
<td>423</td>
<td>VSX</td>
<td>xscvsxddp VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00004E0</td>
<td></td>
<td></td>
<td>423</td>
<td>VSX</td>
<td>xscvsxsp VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00005A0</td>
<td></td>
<td></td>
<td>424</td>
<td>VSX</td>
<td>xscvxuddp VSX Scalar Convert Unsigned Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00004A0</td>
<td></td>
<td></td>
<td>424</td>
<td>VSX</td>
<td>xscvxsp VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00001C0</td>
<td></td>
<td></td>
<td>425</td>
<td>VSX</td>
<td>xsdivdp VSX Scalar Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00001C0</td>
<td></td>
<td></td>
<td>427</td>
<td>VSX</td>
<td>xsdivsp VSX Scalar Divide Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000108</td>
<td></td>
<td></td>
<td>429</td>
<td>VSX</td>
<td>xsmaddadp VSX Scalar Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000008</td>
<td></td>
<td></td>
<td>432</td>
<td>VSX</td>
<td>xsmaddasp VSX Scalar Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000148</td>
<td></td>
<td></td>
<td>429</td>
<td>VSX</td>
<td>xsmaddmdp VSX Scalar Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000048</td>
<td></td>
<td></td>
<td>432</td>
<td>VSX</td>
<td>xsmaddmsp VSX Scalar Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000500</td>
<td></td>
<td></td>
<td>435</td>
<td>VSX</td>
<td>xsmaxd VSX Scalar Maximum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000540</td>
<td></td>
<td></td>
<td>437</td>
<td>VSX</td>
<td>xsmind VSX Scalar Minimum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000188</td>
<td></td>
<td></td>
<td>439</td>
<td>VSX</td>
<td>xsmsubadp VSX Scalar Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000088</td>
<td></td>
<td></td>
<td>442</td>
<td>VSX</td>
<td>xsmsubasp VSX Scalar Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00001C8</td>
<td></td>
<td></td>
<td>439</td>
<td>VSX</td>
<td>xsmsubmdp VSX Scalar Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00000C8</td>
<td></td>
<td></td>
<td>442</td>
<td>VSX</td>
<td>xsmsubmsp VSX Scalar Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000180</td>
<td></td>
<td></td>
<td>445</td>
<td>VSX</td>
<td>xsmulp VSX Scalar Multiply Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000080</td>
<td></td>
<td></td>
<td>447</td>
<td>VSX</td>
<td>xsmuls VSX Scalar Multiply Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00005A4</td>
<td></td>
<td></td>
<td>449</td>
<td>VSX</td>
<td>xsnabsd VSX Scalar Negative Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00005E4</td>
<td></td>
<td></td>
<td>449</td>
<td>VSX</td>
<td>xsnegd VSX Scalar Negate Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000508</td>
<td></td>
<td></td>
<td>450</td>
<td>VSX</td>
<td>xsnmaddad VSX Scalar Negative Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000408</td>
<td></td>
<td></td>
<td>455</td>
<td>VSX</td>
<td>xsnmaddasp VSX Scalar Negative Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000548</td>
<td></td>
<td></td>
<td>450</td>
<td>VSX</td>
<td>xsnmaddmdp VSX Scalar Negative Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000448</td>
<td></td>
<td></td>
<td>455</td>
<td>VSX</td>
<td>xsnmaddmsp VSX Scalar Negative Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000588</td>
<td></td>
<td></td>
<td>458</td>
<td>VSX</td>
<td>xsnmsubad VSX Scalar Negative Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000488</td>
<td></td>
<td></td>
<td>461</td>
<td>VSX</td>
<td>xsnmsubasp VSX Scalar Negative Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00005C8</td>
<td></td>
<td></td>
<td>458</td>
<td>VSX</td>
<td>xsnmsubmdp VSX Scalar Negative Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00004C8</td>
<td></td>
<td></td>
<td>461</td>
<td>VSX</td>
<td>xsnmsubmsp VSX Scalar Negative Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000124</td>
<td></td>
<td></td>
<td>464</td>
<td>VSX</td>
<td>xsrdpi VSX Scalar Round to Double-Precision Integer</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00001AC</td>
<td></td>
<td></td>
<td>465</td>
<td>VSX</td>
<td>xsrdpic VSX Scalar Round to Double-Precision Integer using Current rounding mode</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00001E4</td>
<td></td>
<td></td>
<td>466</td>
<td>VSX</td>
<td>xsrdpim VSX Scalar Round to Double-Precision Integer toward -Infinity</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00001A4</td>
<td></td>
<td></td>
<td>466</td>
<td>VSX</td>
<td>xsrdpip</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000164</td>
<td></td>
<td></td>
<td>467</td>
<td>VSX</td>
<td>xsrdpiz</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000168</td>
<td></td>
<td></td>
<td>468</td>
<td>VSX</td>
<td>xsredp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000068</td>
<td></td>
<td></td>
<td>469</td>
<td>VSX</td>
<td>xsresp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000464</td>
<td></td>
<td></td>
<td>470</td>
<td>VSX</td>
<td>xsrsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000128</td>
<td></td>
<td></td>
<td>471</td>
<td>VSX</td>
<td>xsrsqrdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000028</td>
<td></td>
<td></td>
<td>472</td>
<td>VSX</td>
<td>xsrqntsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000012C</td>
<td></td>
<td></td>
<td>473</td>
<td>VSX</td>
<td>xssqrtdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F000002C</td>
<td></td>
<td></td>
<td>474</td>
<td>VSX</td>
<td>xssqrtp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000140</td>
<td></td>
<td></td>
<td>475</td>
<td>VSX</td>
<td>xssubdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000040</td>
<td></td>
<td></td>
<td>477</td>
<td>VSX</td>
<td>xssubsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000128</td>
<td></td>
<td></td>
<td>479</td>
<td>VSX</td>
<td>xstdivdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00001A8</td>
<td></td>
<td></td>
<td>480</td>
<td>VSX</td>
<td>xstsqrtdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000764</td>
<td></td>
<td></td>
<td>480</td>
<td>VSX</td>
<td>xvabsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000664</td>
<td></td>
<td></td>
<td>481</td>
<td>VSX</td>
<td>xvabssp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000300</td>
<td></td>
<td></td>
<td>482</td>
<td>VSX</td>
<td>xvadddp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000200</td>
<td></td>
<td></td>
<td>486</td>
<td>VSX</td>
<td>xvaddsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000318</td>
<td></td>
<td></td>
<td>488</td>
<td>VSX</td>
<td>xvcmpedqdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000718</td>
<td></td>
<td></td>
<td>488</td>
<td>VSX</td>
<td>xvcmpedqdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000218</td>
<td></td>
<td></td>
<td>489</td>
<td>VSX</td>
<td>xvcmeqsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000618</td>
<td></td>
<td></td>
<td>489</td>
<td>VSX</td>
<td>xvcmeqsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000398</td>
<td></td>
<td></td>
<td>490</td>
<td>VSX</td>
<td>xvcmpgedp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000798</td>
<td></td>
<td></td>
<td>490</td>
<td>VSX</td>
<td>xvcmpgedp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000298</td>
<td></td>
<td></td>
<td>491</td>
<td>VSX</td>
<td>xvcmpegsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000698</td>
<td></td>
<td></td>
<td>491</td>
<td>VSX</td>
<td>xvcmpegsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000358</td>
<td></td>
<td></td>
<td>492</td>
<td>VSX</td>
<td>xvcmpgtdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000758</td>
<td></td>
<td></td>
<td>492</td>
<td>VSX</td>
<td>xvcmpgtdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000258</td>
<td></td>
<td></td>
<td>493</td>
<td>VSX</td>
<td>xvcmpgsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000658</td>
<td></td>
<td></td>
<td>493</td>
<td>VSX</td>
<td>xvcmpgsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000780</td>
<td></td>
<td></td>
<td>494</td>
<td>VSX</td>
<td>xvcpsgndp</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000680</td>
<td></td>
<td></td>
<td>494</td>
<td>VSX</td>
<td>xvcpsgndp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000624</td>
<td></td>
<td></td>
<td>495</td>
<td>VSX</td>
<td>xvcvdsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000760</td>
<td></td>
<td></td>
<td>496</td>
<td>VSX</td>
<td>xvcvdpsxds</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000360</td>
<td></td>
<td></td>
<td>498</td>
<td>VSX</td>
<td>xvcvdpsxws</td>
</tr>
</tbody>
</table>

1480 Power ISA™ - Book Appendices
<table>
<thead>
<tr>
<th>Format</th>
<th>Opcode</th>
<th>Mode Dep.</th>
<th>Privilege</th>
<th>Page</th>
<th>Category</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX2</td>
<td>60 F0000720</td>
<td></td>
<td></td>
<td>500</td>
<td>VSX</td>
<td>xvcvdpuxds</td>
<td>VSX Vector Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000320</td>
<td></td>
<td></td>
<td>502</td>
<td>VSX</td>
<td>xvcvdpuxws</td>
<td>VSX Vector Convert Double-Precision to Unsigned Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000724</td>
<td></td>
<td></td>
<td>504</td>
<td>VSX</td>
<td>xvcvspdp</td>
<td>VSX Vector Convert Single-Precision to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000660</td>
<td></td>
<td></td>
<td>505</td>
<td>VSX</td>
<td>xvcvspxsds</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000260</td>
<td></td>
<td></td>
<td>507</td>
<td>VSX</td>
<td>xvcvspxws</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000620</td>
<td></td>
<td></td>
<td>509</td>
<td>VSX</td>
<td>xvcvpxds</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F0000220</td>
<td></td>
<td></td>
<td>511</td>
<td>VSX</td>
<td>xvcvpxws</td>
<td>VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007E0</td>
<td></td>
<td></td>
<td>513</td>
<td>VSX</td>
<td>xvcvxdsp</td>
<td>VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006E0</td>
<td></td>
<td></td>
<td>513</td>
<td>VSX</td>
<td>xvcvxdsp</td>
<td>VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00003E0</td>
<td></td>
<td></td>
<td>514</td>
<td>VSX</td>
<td>xvcvxdsp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002E0</td>
<td></td>
<td></td>
<td>514</td>
<td>VSX</td>
<td>xvcvxdsp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00007A0</td>
<td></td>
<td></td>
<td>515</td>
<td>VSX</td>
<td>xvcvuxssp</td>
<td>VSX Vector Convert Unsigned Fixed-Point Doubleword to Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00006A0</td>
<td></td>
<td></td>
<td>515</td>
<td>VSX</td>
<td>xvcvuxisp</td>
<td>VSX Vector Convert Unsigned Fixed-Point Doubleword to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003A0</td>
<td></td>
<td></td>
<td>516</td>
<td>VSX</td>
<td>xvcvuxisp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002A0</td>
<td></td>
<td></td>
<td>516</td>
<td>VSX</td>
<td>xvcvuxisp</td>
<td>VSX Vector Convert Signed Fixed-Point Word to Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003C0</td>
<td></td>
<td></td>
<td>517</td>
<td>VSX</td>
<td>xvdidvdp</td>
<td>VSX Vector Divide Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002C0</td>
<td></td>
<td></td>
<td>519</td>
<td>VSX</td>
<td>xvdidisp</td>
<td>VSX Vector Divide Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000308</td>
<td></td>
<td></td>
<td>521</td>
<td>VSX</td>
<td>xvmaddadp</td>
<td>VSX Vector Multiply-Add Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000208</td>
<td></td>
<td></td>
<td>521</td>
<td>VSX</td>
<td>xvmaddasp</td>
<td>VSX Vector Multiply-Add Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000348</td>
<td></td>
<td></td>
<td>524</td>
<td>VSX</td>
<td>xvmaddmdp</td>
<td>VSX Vector Multiply-Add Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000248</td>
<td></td>
<td></td>
<td>524</td>
<td>VSX</td>
<td>xvmaddmdsp</td>
<td>VSX Vector Multiply-Add Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000700</td>
<td></td>
<td></td>
<td>527</td>
<td>VSX</td>
<td>xvmmaxdp</td>
<td>VSX Vector Maximum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000600</td>
<td></td>
<td></td>
<td>529</td>
<td>VSX</td>
<td>xvmmaxsp</td>
<td>VSX Vector Maximum Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000740</td>
<td></td>
<td></td>
<td>531</td>
<td>VSX</td>
<td>xvmindp</td>
<td>VSX Vector Minimum Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000640</td>
<td></td>
<td></td>
<td>533</td>
<td>VSX</td>
<td>xvmindsp</td>
<td>VSX Vector Minimum Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000388</td>
<td></td>
<td></td>
<td>535</td>
<td>VSX</td>
<td>xvmsubadp</td>
<td>VSX Vector Multiply-Subtract Type-A Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000288</td>
<td></td>
<td></td>
<td>535</td>
<td>VSX</td>
<td>xvmsubasp</td>
<td>VSX Vector Multiply-Subtract Type-A Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00003C8</td>
<td></td>
<td></td>
<td>538</td>
<td>VSX</td>
<td>xvmsubmdp</td>
<td>VSX Vector Multiply-Subtract Type-M Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F00002C8</td>
<td></td>
<td></td>
<td>538</td>
<td>VSX</td>
<td>xvmsubmdsp</td>
<td>VSX Vector Multiply-Subtract Type-M Single-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000380</td>
<td></td>
<td></td>
<td>541</td>
<td>VSX</td>
<td>xvmuldsp</td>
<td>VSX Vector Multiply Double-Precision</td>
</tr>
<tr>
<td>XX3</td>
<td>60 F0000280</td>
<td></td>
<td></td>
<td>543</td>
<td>VSX</td>
<td>xvmulsdp</td>
<td>VSX Vector Multiply Single-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00007A4</td>
<td></td>
<td></td>
<td>545</td>
<td>VSX</td>
<td>xvnabsdp</td>
<td>VSX Vector Negative Absolute Value Double-Precision</td>
</tr>
<tr>
<td>XX2</td>
<td>60 F00006A4</td>
<td></td>
<td></td>
<td>545</td>
<td>VSX</td>
<td>xvnabssp</td>
<td>VSX Vector Negative Absolute Value Single-Precision</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Base (hex)</td>
<td>Mode Dep.</td>
<td>Privilege</td>
<td>Page</td>
<td>Category</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00007E4</td>
<td></td>
<td></td>
<td>546</td>
<td>VSX</td>
<td>xvnegdp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00006E4</td>
<td></td>
<td></td>
<td>546</td>
<td>VSX</td>
<td>xvnegsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000708</td>
<td></td>
<td></td>
<td>547</td>
<td>VSX</td>
<td>xvnmadadp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000608</td>
<td></td>
<td></td>
<td>547</td>
<td>VSX</td>
<td>xvnmadasp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000748</td>
<td></td>
<td></td>
<td>552</td>
<td>VSX</td>
<td>xvnmadmdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000648</td>
<td></td>
<td></td>
<td>552</td>
<td>VSX</td>
<td>xvnmadmsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000788</td>
<td></td>
<td></td>
<td>555</td>
<td>VSX</td>
<td>xvnmsubadp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000688</td>
<td></td>
<td></td>
<td>555</td>
<td>VSX</td>
<td>xvnmsubasp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00007C8</td>
<td></td>
<td></td>
<td>558</td>
<td>VSX</td>
<td>xvnmsubmdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00006C8</td>
<td></td>
<td></td>
<td>558</td>
<td>VSX</td>
<td>xvnmsubmisp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000324</td>
<td></td>
<td></td>
<td>561</td>
<td>VSX</td>
<td>xvrdpi</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00003AC</td>
<td></td>
<td></td>
<td>561</td>
<td>VSX</td>
<td>xvrdpic</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00003E4</td>
<td></td>
<td></td>
<td>562</td>
<td>VSX</td>
<td>xvrdpim</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00003A4</td>
<td></td>
<td></td>
<td>562</td>
<td>VSX</td>
<td>xvrdpip</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000364</td>
<td></td>
<td></td>
<td>563</td>
<td>VSX</td>
<td>xvrdpiz</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000368</td>
<td></td>
<td></td>
<td>564</td>
<td>VSX</td>
<td>xvrredp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000268</td>
<td></td>
<td></td>
<td>565</td>
<td>VSX</td>
<td>xvresp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000224</td>
<td></td>
<td></td>
<td>566</td>
<td>VSX</td>
<td>xvrspi</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00002AC</td>
<td></td>
<td></td>
<td>566</td>
<td>VSX</td>
<td>xvrspic</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00002E4</td>
<td></td>
<td></td>
<td>567</td>
<td>VSX</td>
<td>xvrspim</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00002A4</td>
<td></td>
<td></td>
<td>567</td>
<td>VSX</td>
<td>xvrspip</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000264</td>
<td></td>
<td></td>
<td>568</td>
<td>VSX</td>
<td>xvrspiz</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000328</td>
<td></td>
<td></td>
<td>568</td>
<td>VSX</td>
<td>xvrsqrtedp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F0000228</td>
<td></td>
<td></td>
<td>570</td>
<td>VSX</td>
<td>xvrsqrtesp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F000032C</td>
<td></td>
<td></td>
<td>571</td>
<td>VSX</td>
<td>xvsqrtddp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F000022C</td>
<td></td>
<td></td>
<td>572</td>
<td>VSX</td>
<td>xvsqrtsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000340</td>
<td></td>
<td></td>
<td>573</td>
<td>VSX</td>
<td>xsubitdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000240</td>
<td></td>
<td></td>
<td>575</td>
<td>VSX</td>
<td>xsubsbp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00003E8</td>
<td></td>
<td></td>
<td>577</td>
<td>VSX</td>
<td>xvtdivdp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00002E8</td>
<td></td>
<td></td>
<td>578</td>
<td>VSX</td>
<td>xvtdivsp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00003A8</td>
<td></td>
<td></td>
<td>579</td>
<td>VSX</td>
<td>xvtsqrtddp</td>
</tr>
<tr>
<td>XX2</td>
<td>60</td>
<td>F00002A8</td>
<td></td>
<td></td>
<td>579</td>
<td>VSX</td>
<td>xvtsqrtsp</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000410</td>
<td></td>
<td></td>
<td>580</td>
<td>VSX</td>
<td>xxland</td>
</tr>
<tr>
<td>Format</td>
<td>Primary</td>
<td>Base (hex)</td>
<td>Mode Dep. 1</td>
<td>Privilege 1</td>
<td>Page</td>
<td>Category 1</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>------------</td>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000450</td>
<td>580</td>
<td>VSX</td>
<td></td>
<td>VSX Logical AND with Complement</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00005D0</td>
<td>581</td>
<td>VSX</td>
<td></td>
<td>VSX Logical Equivalence</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000590</td>
<td>581</td>
<td>VSX</td>
<td></td>
<td>VSX Logical NAND</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000510</td>
<td>582</td>
<td>VSX</td>
<td></td>
<td>VSX Logical NOR</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000490</td>
<td>583</td>
<td>VSX</td>
<td></td>
<td>VSX Logical OR</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000550</td>
<td>582</td>
<td>VSX</td>
<td></td>
<td>VSX Logical OR with Complement</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F00004D0</td>
<td>583</td>
<td>VSX</td>
<td></td>
<td>VSX Logical XOR</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000090</td>
<td>584</td>
<td>VSX</td>
<td></td>
<td>VSX Merge High Word</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000190</td>
<td>584</td>
<td>VSX</td>
<td></td>
<td>VSX Merge Low Word</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000050</td>
<td>585</td>
<td>VSX</td>
<td></td>
<td>VSX Permute Doubleword Immediate</td>
<td></td>
</tr>
<tr>
<td>XX4</td>
<td>60</td>
<td>F0000030</td>
<td>585</td>
<td>VSX</td>
<td></td>
<td>VSX Select</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000010</td>
<td>586</td>
<td>VSX</td>
<td></td>
<td>VSX Shift Left Double by Word Immediate</td>
<td></td>
</tr>
<tr>
<td>XX3</td>
<td>60</td>
<td>F0000290</td>
<td>586</td>
<td>VSX</td>
<td></td>
<td>VSX Splat Word</td>
<td></td>
</tr>
</tbody>
</table>

1 See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.
Mode Dependency and Privilege Abbreviations

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

Key to Mode Dependency Column

<table>
<thead>
<tr>
<th>Mode Dep.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT</td>
<td>If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.</td>
</tr>
<tr>
<td>SR</td>
<td>The setting of status registers (such as XER and CR0) is mode-dependent.</td>
</tr>
<tr>
<td>32</td>
<td>The instruction can be executed only in 32-bit mode.</td>
</tr>
<tr>
<td>64</td>
<td>The instruction can be executed only in 64-bit mode.</td>
</tr>
</tbody>
</table>

Key to Privilege Column

<table>
<thead>
<tr>
<th>Priv.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Denotes a privileged instruction.</td>
</tr>
<tr>
<td>O</td>
<td>Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspri), depending on the SPR or PMR number.</td>
</tr>
<tr>
<td>H</td>
<td>Denotes an instruction that can be executed only in hypervisor state &lt;S,E,HV&gt;</td>
</tr>
<tr>
<td>PH</td>
<td>Denotes a hypervisor privileged instruction if Category Embedded.Hypervisor is implemented; otherwise denotes a privileged instruction.</td>
</tr>
<tr>
<td>M</td>
<td>Denotes an instruction that is treated as privileged or nonprivileged, depending on the value of the UCLE bit in the MSR.</td>
</tr>
</tbody>
</table>
Index

Numerics
2 844
3 844
32-bit mode 891

A
a bit 34
AA field 17
address 23
effective 26
effective address 889, 1073
real 889, 1076
address compare 889, 955, 963
address translation 904, 1084
32-bit mode 891
EA to VA 891
esid to vsid 891
overview 895
PTE
page table entry 900, 904, 1089
Reference bit 904
RPN
real page number 898
VA to RA 898
VPN
virtual page number 898
address wrap 889, 1076
addresses
accessed by processor 895
implicit accesses 895
interrupt vectors 895
with defined uses 895
addressing mode
D-mode 1267
A-form 16
aliasing 742
alignment
effect on performance 753, 979, 983, 1003, 1213
Alignment interrupt 958, 1017, 1170
assembler language
extended mnemonics 709, 1015, 1245
mnemonics 709, 1015, 1245
symbols 709, 1015, 1245
atomic operation 744
atomicity 737
single-copy 737
Auxiliary Processor 4
Auxiliary Processor Unavailable interrupt 1173

B
BA field 17
BA instruction field 1263
BB field 17
BC field 17
BD field 18
BD instruction field 1263
BE
See Machine State Register
BF field 18
BF instruction field 1264
BFA field 18
BFA instruction field 1264
B-form 14
BH field 18
BI field 18, 20
block 736
BO field 18, 34
boundedly undefined 4
Bridge 925
Segment Registers 925
SR 926
brinc 594
BT field 18
bytes 3

C
C 117
CA 46, 804
cache management instructions 761
cache model 737
cache parameters 759
Caching Inhibited 738
Change bit 904
CIA 7
Come-From Address Register 881, 1361
consistency 742
count
definition 839, 1024
synchronization 841, 1026
Control Register 872
Count Register 881, 1050, 1274, 1361
CR 30
Critical Input interrupt 1165
Critical Save/Restore Register 1 1147
CSRR1 1147
CTR 32, 1274
CTRL
See Control Register
Current Instruction Address 863, 1040, 1278

D
d field 18
d instruction field 1264
DABR interrupt 980
DABR(X)
See Data Breakpoint Register (Extension)
DAR
See Data Address Register
Data 1066
data access 889, 1076
Data Address Breakpoint Register (Extension) 854, 980, 1010
data address compare 955, 963
Data Address Register 881, 939, 940, 956, 959, 964, 1361
data cache instructions 60, 763
Data Exception Address Register 1148
data exception address register 1148
Alignment interrupt 1017
Data TLB Error interrupt 1176
DC 803
dcbf instruction 773
dcbt instruction 751, 773, 955, 963
dcbt instruction 770, 1063, 1122
dctbts 1123
dctbts instruction 771, 1066, 1122
dcbz instruction 772, 917, 955, 959, 963, 1017, 1067, 1118
DEAR 1148
Debug Interrupt 1178
DEC
See Decrementer
decimal carries 803
Decrementer 881, 974, 975, 1050, 1199, 1200, 1208, 1361
Decrementer Interrupt 1173
Decrementer interrupt 887, 961
defined instructions 21
denormalization 121, 331
denormalized number 120, 330
D-form 14
D-mode addressing mode 1267
double-precision 121
doublewords 3
DQ field 18
DQ-form 14
DR
See Machine State Register
DS 804
DS field 18
DS-form 14
DSISR
See Data Storage Interrupt Status Register
E
E (Enable bit) 1001
EA 26
eicix instruction 825, 826, 955, 958, 963, 1001
eicow instruction 825, 826, 955, 958, 963, 1001
EE
See Machine State Register
effective address 26, 889, 895, 1073
size 891
translation 896
Effective Address Overflow 963
eieio instruction 742, 790, 1092
emulation assist 840, 1025
Endianness 740
EQ 30, 31, 804
ESR 1150, 1151
evabs 594
evaddiw 594
evaddsmsiaaw 594
evaddssiaaw 595
evlwhex 603
exception 1145
alignment exception 1170
critical input exception 1165
data storage exception 1166
e external input exception 1170
illegal instruction exception 1171
instruction storage exception 1168
instruction TLB miss exception 1177
machine check exception 1165
privileged instruction exception 1172
program exception 1171
system call exception 1173, 1183, 1184
trap exception 1172
exception priorities 1190
system call instruction 1191
trap instructions 1191
Exception Syndrome Register 1150, 1151
exception syndrome register 1150, 1151
exception vector prefix register 1148, 1149
Exceptions 1145
exceptions
address compare 889, 955, 963
definition 839, 1024
Effective Address Overflow 963
page fault 889, 903, 955, 963, 1075
protection 889, 1075
segment fault 889
storage 889, 1075
execution synchronization 842, 1026
extended mnemonics 827
External Access Register 881, 955, 963, 1001, 1010, 1050, 1362
External Control 825
External Control instructions
eciwx 826
ecowx 826
External Input interrupt 1170
External interrupt 887, 958
F
FE 31, 118, 325
FE0
See Machine State Register
FE1
See Machine State Register
FEX 117
FG 31, 118, 325
FI 117
Fixed-Interval Timer interrupt 1174
Fixed-Point Exception Register 881, 1050, 1361
FL 31, 118, 325
FLM field 18
floating-point
denormalization 121, 331
double-precision 121
exceptions 115, 124, 328
inexact 128, 356
invalid operation 126, 343
overflow 127, 351
underflow 128, 353
zero divide 126, 349
evaluation models 125, 327
invalid numbers 121, 331
denormalized 120, 330
not a number 120, 330
zero 120, 330
rounding 123, 335
sign 121, 331
single-precision 121
Floating-Point Unavailable interrupt 961, 966, 967, 1172
forward progress 747
FP
See Machine State Register
FPCCR 117, 325
FPR 116
FPRF 117
FPSCR 116, 323
C 117
FE 118, 325
FEX 117
FG 118, 325
FI 117
FL 118, 325
FPCC 117, 325
FPKF 117
FR 117
FU 118
FX 116, 323
NI 118, 326
OE 118, 326
OX 117, 323
RN 118, 326
UE 118, 326
UX 117
VE 118, 325
VX 117, 323
VXCVI 118, 325
VXIDI 117, 324
VXIMZ 117
VXISI 117, 324
VXSNAI 117, 324
VXSOFT 118
VXSORT 118, 325
VXVC 117
VXZDZ 117, 324
XE 118
XX 117
ZE 118
ZX 117
FR 117
FRA field 18, 19
FRB field 18
FRS field 18
FRT field 19
FU 31, 118
FX 116, 323
FXCC 804
FXM field 19
FXM instruction field 1264

G

GPR 45
GT 30, 31, 804
Guarded 739

H

halfwords 3
hardware
definition 840, 1025
hardware description language 6
hashed page table
size 901
HDEC
See Hypervisor Decrementer
HDICE
See Logical Partitioning Control Register
HEIR
See Hypervisor Emulated Instruction Register
hrfid instruction 857, 971
HRMOR
See Hypervisor Real Mode Offset Register
HSPRGN
See software-use SPRs
HTABORG 901
HTABSIZE 901
HV
See Machine State Register
hypervisor 843, 1031
Hypervisor Decrementer 881, 975, 1010, 1362
Hypervisor Decrementer interrupt 961
Hypervisor Emulated Instruction Register 882, 940, 1362
Hypervisor Machine Status Save Restore Register
See HSRR0, HSRR1
Hypervisor Machine Status Save Restore Register 0 939
Hypervisor Real Mode Offset Register 757, 846, 847, 1010

I

icbi instruction 751, 762, 955, 963
icbt instruction 762
I-form 14
ILE
See Logical Partitioning Control Register
illegal instructions 21
implicit branch 889, 1075
imprecise interrupt 946, 1157
inexact 128, 356
infinity 120, 330
in-order operations 890, 1076
instruction 955, 963

field
BA 1263
BD 1263
BF 1264
BFA 1264
D 1264
FXM 1264
L 1264
LK 1264
Rc 1264
SH 1264, 1269
SI 1264
UL 1264
WS 1265

fields
AA 17
BA 17
BB 17
BC 17
BD 18
BF 18
BFA 18
BH 18
BI 18, 20
BO 18
BT 18
D 18
DQ 18
DS 18
FLM 18
FRA 18, 19
FRB 18
FRC 18
FRS 18
FRT 19
FXM 19
L 19
LEV 19
LI 19
LK 19
MB 19
ME 19
NB 19
OE 19
PMRN 19
RA 19
RB 19, 20
Rc 19
RS 20
RT 20
SH 20
SI 20
SPR 20
SR 20
TBR 20
TH 20
TO 20
U 20
UI 20

formats
A-form 16
B-form 14
D-form 14
DQ-form 14
DS-form 14
I-form 14
MD-form 16
MDS-form 16
M-form 16
SC-form 14
VA-form 16
VX-form 17
XFL-form 16
X-form 15
XFX-form 15
XL-form 15
XO-form 16
XS-form 16

interrupt control 1278
mtmsr 1055
partially executed 1186
rfci 1280

instruction cache instructions 762
instruction fetch 889, 1075
effective address 889, 1075
implicit branch 889, 1075
Instruction Fields 1263
instruction restart 755
Instruction Segment interrupt 957, 965
instruction storage 735
Instruction Storage interrupt 957, 1168
Instruction TLB Error Interrupt 1177
instruction-caused interrupt 946

Instructions
brnc 594
dcbts 1123
evabs 594
evaddiw 594
evaddsmiaaw 594
evaddssiaaw 595
evldhex 603
instructions
classes 21
dcbx 770, 1118
dcbtx 771, 773, 955, 963
dcbt 770, 1063, 1122
dcbx771, 1066, 1122
dcbz 772, 917, 959, 1017, 1067, 1118
defined 21
forms 22
eciwx 825, 826, 955, 958, 963, 1001
ecowx 825, 826, 955, 958, 963, 1001
eieio 742, 790, 1092
hrfhd 857, 971
icbti 751, 762, 955, 963
icbi 762
illegal 21
invalid forms 22
isync 751, 776
ldarx 744, 782, 784, 955, 958, 963
lmw 958
lookaside buffer 918
lq 58, 958
lwa 959
lwarx 744, 777, 778, 955, 958, 963
lwaux 959
lwsync 786
mbar 790
mfmsr 857, 888, 1056
mfspr 885, 1054
mfsr 927
mfsrin 927
mtftb 813
mtmsr 857, 886, 971
mtmsrd 857, 887, 971
address wrap 889, 1076
mtspr 884, 1063
msr 926
mtsrin 926
optional
See optional instructions
preferred forms 22
ptlesync 786, 842, 1092
reserved 21
rfci 1041
rfid 751, 857, 864, 865, 949, 971
rfnc 1042
rfscv 971
sc 822, 823, 824, 863, 867, 962, 1040
sblia 920, 923
sbie 919
sblmfee 923
sblmfiv 922
sblmte 921
stdcx. 744, 955, 958, 963
stmw 958
storage control 759, 795, 917, 1118
stq 58, 958
stw 1017
stwcx. 744, 780, 781, 782, 785, 955, 958, 963
stwx 1017
sync 751, 786, 842, 904
tbia 904, 933
tbie 904, 928, 934, 936, 1093
tbile 931
tbsync 934, 1092
wrtex 1056
wrtexl 1057
interrupt 1145
Alignment 958, 1017
alignment interrupt 1170
DABR 980
Data Segment 956
Data Storage 955, 963
data storage interrupt 1166
Decrementer 887, 961
definition 839, 1024, 1025
External 887, 958
external input interrupt 1170
Floating-Point Unavailable 961, 966, 967
Hypervisor Decrementer 961
imprecise 946, 1157
instruction
partially executed 1186
Instruction Segment 957, 965
Instruction Storage 957, 1168
instruction storage interrupt 1168
instruction TLB miss interrupt 1177
**K**
- K bits 908
- key, storage 908

**L**
- dcbf 955, 963
- instructions
  - dcbf 955, 963
- L field 19
- L instruction field 1264
- language used for instruction operation description 6
- Idax instruction 744, 782, 784, 955, 958, 963
- LE
  - See Machine State Register
- LEV field 19
- Li field 19
- Link Register 881, 1050, 1274, 1361
- LK field 19
- LK instruction field 1264
- lmw instruction 958
- Logical Partition Identification Register 847
- Logical Partitioning 843, 1031
- Logical Partitioning Control Register 759, 843, 882, 918, 1010, 1362
- HDICE Hypervisor Decrementer Interrupt Conditionally Enable 846, 854, 886, 887, 961, 962, 1011
- ILE Interrupt Little-Endian 844, 951
- ISL Ignore Large Page Specification 844
- ISL Ignore SLB Large Page Specification 844
- LPES Logical Partitioning Environment Selector 846, 854
- RMI Real Mode Caching Inhibited Bit 846
- RMLS Real Mode Offset Selector 843, 844, 1013
- VC 1013
- VC Virtualization Control 843
- VPM Virtualized Partition Memory 843
- VRMASD 1013
- VRMASD Virtual Real Mode Area Segment Descriptor 844
- lookaside buffer 918
- LPAR (see Logical Partitioning) 843, 1031
- LPCR
  - See Logical Partitioning Control Register
- LPES
  - See Logical Partitioning Control Register
- LPIDR
  - See Logical Partition Identification Register
- Iq instruction 58, 958
- LR 32, 1274
- LT 30, 31, 804
- Iwa instruction 959
- IwaX instruction 744, 777, 778, 955, 958, 963
- Iwax instruction 959
- IwXsync instruction 786

**M**
- Machine 1035
- Machine Check 1157
- Machine Check Interrupt 953, 1165
- Machine State Register 857, 863, 886, 887, 888, 947, 949, 950, 1035, 1056
- BEBranch Trace Enable 859
- DRRData Relocate 859
- EEE External Interrupt Enable 858, 886, 887
- FE0FP Exception Mode 858
- FE1FP Exception Mode 859
- FFPF Available 858
- HVHypervisor State 857
- IRInstruction Relocate 859
- LE Little-Endian Mode 859
- MEM Machine Check Enable 858
- PMMP Performance Monitor Mark 859
- PR Problem State 858
- RR Recoverable Interrupt 859, 886, 887

**Index** 1489
SESingle-Step Trace Enable 858
SFSixty Four Bit mode 759, 760, 857, 858, 889, 1076
VECAvailable 858
Machine Status Save Restore Register
See SRR0, SRR1
Machine Status Save Restore Register 0 939, 947, 949
Machine Status Save Restore Register 1 947, 949, 960
main storage 735
MB field 19
mbar instruction 790
MD-form 16
MDS-form 16
ME
See Machine State Register
ME field 19
memory barrier 742
Memory Coherence Required 739
mfspr instruction 885, 1054
mfsr instruction 927
mfsrin instruction 927
mftb instruction 813
M-form 16
mfsr instruction 927
mfsrin instruction 927
mftb instruction 813
Mnemonics 1261
mnemonics
extended 709, 1015, 1245
mode change 889, 1076
move to machine state register 1055
MSR
See Machine State Register
mtmsr 1055
mtmsr instruction 857, 886, 971
mtmsrd instruction 857, 887, 971
mtspr instruction 857, 886, 971
mtsr instruction 926
mtsr instruction 926
N
NB field 19
Next Instruction Address 863, 864, 1040, 1041, 1042, 1043, 1278, 1281
NI 118, 326
NIA 7
no-op 82
normalization 121, 331
normalized number 120, 330
not a number 120, 330
O
OE 118, 326
OE field 19
optional instructions 918
slbia 920, 923
slbie 919
tbia 933
tbie 928
tbile 931
tbsync 934
out-of-order operations 890, 1076
OV 45, 804
overflow 127, 351
OX 117, 323
P
page 736
size 891
page fault 889, 903, 955, 963, 1075
page table
search 902
update 1092
page table entry 900, 904, 1089
<table>
<thead>
<tr>
<th>SPRGn</th>
<th>General Purpose Registers 45</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPR</td>
<td>Program Priority Register 881, 1052, 1054</td>
</tr>
<tr>
<td>SRR0</td>
<td>Machine Status Save Restore Register 0 939, 947, 949</td>
</tr>
<tr>
<td>SRR1</td>
<td>Machine Status Save Restore Register 1 947, 949, 960</td>
</tr>
<tr>
<td>TB</td>
<td>Time Base 973, 1197</td>
</tr>
<tr>
<td>TBL</td>
<td>Time Base Lower 881, 973, 1050, 1197, 1362</td>
</tr>
<tr>
<td>TBU</td>
<td>Time Base Upper 881, 973, 1050, 1197, 1362</td>
</tr>
<tr>
<td>XER</td>
<td>Fixed-Point Exception Register 857, 860, 881, 1035, 1036, 1037, 1050, 1221, 1222, 1361</td>
</tr>
<tr>
<td>relocation</td>
<td>data 889, 1076</td>
</tr>
<tr>
<td>reserved field</td>
<td>5, 840</td>
</tr>
<tr>
<td>reserved instructions</td>
<td>21</td>
</tr>
<tr>
<td>return from critical interrupt</td>
<td>1280</td>
</tr>
<tr>
<td>rfc1</td>
<td>1280</td>
</tr>
<tr>
<td>rfc1 instruction</td>
<td>1041</td>
</tr>
<tr>
<td>rfd instruction</td>
<td>751, 857, 864, 865, 949, 971</td>
</tr>
<tr>
<td>rfmc1 instruction</td>
<td>1042</td>
</tr>
<tr>
<td>rfscv instruction</td>
<td>971</td>
</tr>
<tr>
<td>Ri</td>
<td>See Machine State Register RId (Resource ID) 1001</td>
</tr>
<tr>
<td>RMOR</td>
<td>See Logical Partitioning Control Register</td>
</tr>
<tr>
<td>RMLS</td>
<td>See Logical Partitioning Control Register</td>
</tr>
<tr>
<td>SRR0</td>
<td>See Real Mode Offset Register</td>
</tr>
<tr>
<td>RN</td>
<td>See Machine State Register</td>
</tr>
<tr>
<td>size 91</td>
<td>segment size 891</td>
</tr>
<tr>
<td>type 91</td>
<td>type 891</td>
</tr>
<tr>
<td>SC-form 14</td>
<td>Segment Lookaside Buffer</td>
</tr>
<tr>
<td>SDR1</td>
<td>See Storage Description Register 1</td>
</tr>
<tr>
<td>SE</td>
<td>See Machine State Register</td>
</tr>
<tr>
<td>segment 29</td>
<td>definition 840, 1024</td>
</tr>
<tr>
<td>SH field 20</td>
<td>sequential execution model</td>
</tr>
<tr>
<td>SH instruction field</td>
<td>1264, 1269</td>
</tr>
<tr>
<td>SI field 20</td>
<td>sign 121, 331</td>
</tr>
<tr>
<td>SI instruction field</td>
<td>1264</td>
</tr>
<tr>
<td>single-copy atomicity 737</td>
<td>single-precision 121</td>
</tr>
<tr>
<td>SLB 896, 918</td>
<td>entry 897</td>
</tr>
<tr>
<td>sibia instruction 920, 923</td>
<td></td>
</tr>
</tbody>
</table>
slbie instruction 919
slbfmfee instruction 923
slbfmfev instruction 922
slbmt instruction 921
SO 30, 31, 45, 803
software-use SPRs 881, 1050, 1361
Special Purpose Registers 880
speculative operations 890, 1076
split field notation 14
SPR field 20
SR 925
SR field 20
SRR0 1145, 1146
SRR1 1145, 1146
stdcx. instruction 744, 955, 958, 963
stmw instruction 958
storage
access order 742
accessed by processor 895
atomic operation 744
attributes
Endianness 740
implicit accesses 895
instruction restart 755
interrupt vectors 895
N 902
No-execute 902
order 742
ordering 742, 786, 790
protection
translation disabled 913
reservation 745
shared 742
with defined uses 895
storage access 735
definitions
program order 735, 736
floating-point 133, 358
storage access ordering 829
storage address 23
storage control
instructions 917, 1118
storage control attributes 738
storage control instructions 759, 795
Storage Description Register 1 881, 901, 1010, 1361, 1362
storage key 908
storage location 735
storage operations
in-order 890, 1076
out-of-order 890, 1076
speculative 890, 1076
storage protection 908
string instruction 1096
TLB management 1124
stq instruction 58, 958
string instruction 1096
stw instruction 1017
stwx. instruction 744, 780, 781, 782, 785, 955, 958, 963
stwx instruction 1017
symbols 709, 1015, 1245
sync instruction 751, 786, 842, 904
synchronization 841, 1026, 1092
count 841, 1026
execution 842, 1026
interrupts 946
Synchronize 742
Synchronous 1156
system call instruction 1191
System Call interrupt 962, 1173, 1183, 1184
System Reset interrupt 952
system-caused interrupt 946

T

T bit 34
table update 1092
TB 813, 817, 821
TBL 813, 817, 821
TBR field 20
TGCC 804
TH field 20
Time Base 813, 817, 821, 973, 1197
Time Base Lower 881, 973, 1050, 1197, 1362
Time Base Upper 881, 973, 1050, 1197, 1362
TLB 904, 918, 1077
TLB management 1124
tibia instruction 904, 933
tble instruction 904, 928, 934, 936, 1093
tbiel instruction 931
tbsync instruction 934, 1092
TO field 20
Translation Lookaside Buffer 1077
translation lookaside buffer 904
trap instructions 1191
trap interrupt
definition 840, 1024

U

U field 20
UE 118, 326
UI field 20
UI instruction field 1264
UMMCRI (user monitor mode control register 1) 1255
undefined 7
boundedly 4
underflow 128, 353
UX 117

V

VA-form 16
VC
See Logical Partitioning Control Register
VE 118, 325
VEC
See Machine State Register
virtual address 895, 898
generation 896
size 891
virtual page number 900, 1089
virtual storage 736
VPM
See Logical Partitioning Control Register
VRMASD
See Logical Partitioning Control Register
VRXCVI
See Logical Partitioning Control Register
VX 117, 323
VXCVI 118, 325
VX-form 17
VXIDI 117, 324
VXIMZ 117
VXISI 117, 324
VXSAN 117, 324
VXSOFI 118
VXSQRT 118, 325
VXVC 117
VXZDZ 117, 324

W

Watchdog Timer interrupt 1175
words 3
Write Through Required 738
wrtei instruction 1056
wrteei instruction 1057
WS instruction field 1265
X
XE 118
XER 45, 857, 860, 1035, 1036, 1037, 1221, 1222
XFL-form 16
X-form 15
XFX-form 15
XL-form 15
XO-form 16
XS-form 16
XX 117

Z
z bit 34
ZE 118
zero 120, 330
zero divide 126, 349
ZX 117
Last Page - End of Document