BluEJAMM: A Bluespec Embedded Java Architecture with Memory Management

Flavius Gruian
Dept. of Computer Science
Lund University
221 00 Lund, Sweden
flavius.gruian@cs.lth.se

Mark Westmijze
Dept. of Computer Science
University of Twente
Enschede, The Netherlands
m.westmijze@student.utwente.nl

Abstract

This paper presents BLUEJAMM*, a prototype architecture suitable for embedded systems based on a Java native processor.* BLUEJEP*, the processor, which is a microprogrammed pipelined stack machine, and its hardware memory management unit were developed in Bluespec SystemVerilog (BSV). A relatively new high-level of abstraction hardware description language, BSV proved to be an excellent choice for rapid prototyping and architecture exploration. The architecture, which has been implemented and tested on a Xilinx FPGA, is currently used to evaluate a number of interesting Java specific techniques, such as runtime bytecode folding and real-time garbage collection.*

1 Introduction

With its attractive features, such as object orientation, runtime safeness, automatic memory management and portability, Java is today popular, not only for desktop applications, but for embedded and even real-time systems. However, desktop Java environments are highly demanding in terms of resources and performance, making them less suitable for small embedded systems. Furthermore, real-time systems require accurate timing and predictability, which is impossible to achieve with typical Java environments due to dynamic class loading and garbage collection.

In this context, a number of embedded and real-time solutions based on Java have appeared in the last few years. Starting from pure software virtual machines [11, 19] and Java-to-C translation [13,14] to processors with direct bytecode execution [1, 6, 16, 18, 20], a vast variety of more or less complete solutions have been presented.

In this paper we describe BLUEJAMM, a Bluespec based Embedded Java Architecture with Memory Management. Our solution has its starting point in the Java Optimized Processor (JOP, [16]), which is a micro-programmed stack machine. However, BLUEJEP, our processor, has a somewhat different architecture [9], including a memory manager, and has been completely written in Bluespec SystemVerilog (BSV [3]). BSV is a rule based, strongly-typed, declarative hardware specification language making use of Term Rewriting Systems [10] to describe computation as atomic state changes. A BSV source can be translated by a BSV compiler to Verilog code, for synthesis with a regular tool flow, or to an executable simulation, for test and debug. Although relatively new, Bluespec seems to have captured the interest of industry and academia, as shown by the BSV designs making their appearance (i.e. [4, 5, 21]).

The paper is organized as follows. Section 2 introduces the BLUEJAMM system architecture and a few possible configurations. Section 3 briefly presents the BLUEJEP processor, while section 4 focuses on the memory management and garbage collection unit. A few experimental results and implementation features make the subject of section 5. Section 6 presents some of the related work and finally, section 7 summarizes our work.

2 System Architecture and Configurations

The BLUEJAMM architecture is a typical system-on-chip, designed for embedded applications, implemented in our case on a FPGA. The simplest and smallest configuration, shown on the left in Figure 1, contains the BLUEJEP processor, a RAM (storing the Java application and heap), a serial port (RS232), a timer, and some general purpose input/output (LEDs and switches), all connected through a system bus (OPB). Memory management is carried out in software in this configuration, using a stop-the-world garbage collection mechanism.

For those applications that can trade-off device area for higher performance, the BLUEJAMM architecture offers

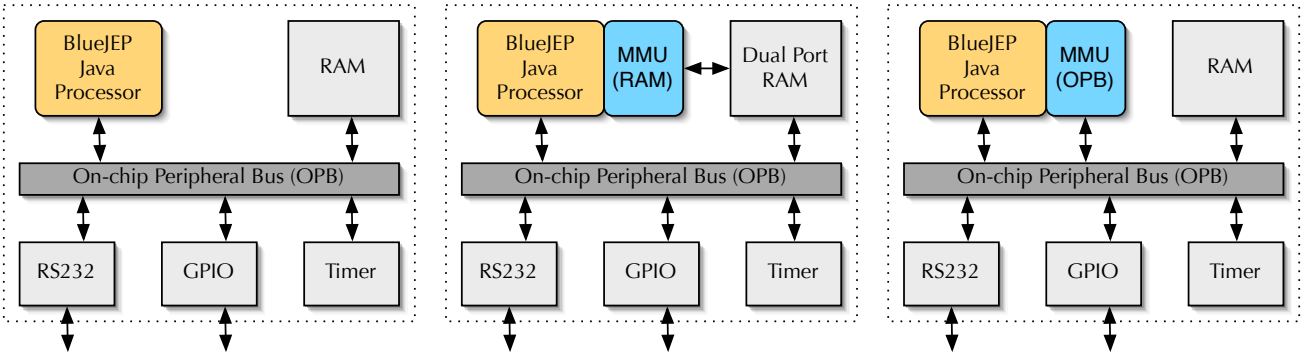

Figure 1. Configurations of the BluEJAMM system architecture. *Left*: **software memory management.** *Middle*: **with MMU when dual port memories are available.** *Right*: **with MMU using the system bus.**

a hardware memory management unit (MMU). The MMU takes care of allocating new objects and carries out the garbage collection much more efficiently than the software counterpart. For this, the MMU needs access to the heap, which is achieved either through a second port, if dual port memories are available (see Figure 1, middle) or through the system bus otherwise (see Figure 1, right). Naturally, the dual-port solution offers higher performance for two reasons. First, the direct memory access is faster than the access through the system bus. In particular for our platform, data access has 1 clock cycle delay for Xilinx block RAMs compared to 3 clock cycles delay on the OPB. Secondly, the MMU does not have to compete with the processor for the same resource in order to carry out the work. This last issue can become a problem especially in the case of concurrent garbage collection. Currently, the MMU supports only stop-the-world garbage collection, but a fully concurrent version is under development, following the ideas first introduced by the authors in [8].

3 BlueJEP, the native Java processor

The processor architecture that we use in system has its roots in the Java optimized processor (JOP [16]), a pipelined micro-programmed processor, able to directly execute bytecodes. The core is a stack machine, in the line with the Java virtual machine, that can execute simple bytecodes as single micro-instructions, while the more complex ones are implemented either as micro-programs or even Java methods. Being designed for embedded and real-time systems, this is not a general Java environment. For instance, class loading is carried out and an executable image (still as bytecodes) is generated offline. Taking a step towards higher-level of abstraction specifications, our processor, BLUEJEP was written in Bluespec System Verilog, as an alternative to VHDL.

3.1 Pipeline Architecture

Initially, when starting on our BSV design, we decided to use the tools already implemented for JOP (micro-assembler and executable image generator), we also wanted to have the same micro-instruction set and micro-code. Nevertheless, as we became more familiar with BSV, we decided that a longer pipeline would be more interesting, more flexible and modular, and hopefully faster. Thus BLUEJEP crystalized into a six stages pipeline (Figure 2), having the micro-instruction set, micro-code and bytecode executable image similar to JOP. Besides the actual pipeline, BLUEJEP features a 256 words stack and a few registers, enumerated in Table 1. Bytecodes are fed into the pipeline with the help of a bytecode cache, which currently stores the code for one method (maximum 1KB), but can be easily modified.

The BLUEJEP processor went through at least three different versions until the solution described in here. Earlier versions would stall the pipeline any time a data or a control hazard would occur, which meant more complex control. The current version only stalls on data hazards, and uses speculative execution on branches, which means simpler control, higher-performance, but wider pipeline registers (for context saving). Whenever an unexpected deviation of control occurs the pipeline is flushed and the execution resumes using the context(JPC, PC, SP) associated with the instruction that caused the branch.

3.1.1 Stage 1: Fetch Bytecode

The *Fetch Bytecode* stage fetches bytes from the bytecode cache and feeds them to the next stage, along with its translation into micro-address (using the *BC2microA* table) and their associated JPC.

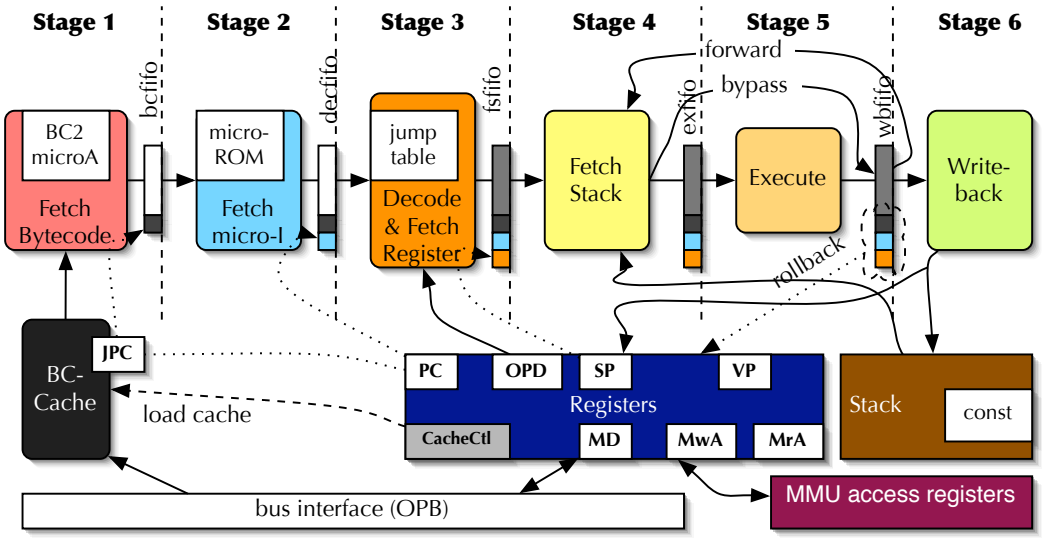

Figure 2. The BlueJEP pipeline architecture

Table 1. BlueJEP registers

Register	Function	Description
PC	micro-program counter	Keeps track of micro-instructions.
JPC	java program counter	Keeps track of bytecodes. Managed by the bytecode cache.
SP	stack pointer	Keeps track of the top of stack.
VP	variable pointer	Points to the base of the local variable frame for the current method.
CACHECTL	cache control register	Writes cause flush and load. Used by method calls and returns.
MWA/MRA	external memory address	Indicates bus address and type of access.
MD	external memory data	Data to write or data read from the given bus address.
OPD	java operand	Operand bytes are fetched in here. Read access on 8/16 bits, signed/unsigned.

3.1.2 Stage 2: Fetch micro-I

The *Fetch micro-I* stage keeps track of the micro-program counter (PC), fetching new micro-instructions from the micro-ROM, and feeds them to the next stage along with their associated PC. Whenever the micro-code for the current bytecode is completed, and the next must be executed, it dequeues a micro-address from the *bcfifo*, and updates the PC accordingly. If the next byte is a java operand rather than a bytecode, it is shifted in the OPD, a 16-bit register.

3.1.3 Stage 3: Decode

The *Decode* stage dequeues and decodes the next micro-instruction from the *fsfifo*. The decoded micro-instructions are either data moving instructions (one source and one destination) or an operation (two sources, one operation, one destination). References (sources and destination) may be immediate values, registers, or stack addresses referred by values or registers. Necessary register values are fetched in this stage, while stack locations are fetched in the next. Along with references and operations, the context information received from the *decfifo* is passed on, augmented with the current value of SP. The *Decode* stage stalls if the contents of a register is required, but about to be changed by an instruction present in the later stages (RAW hazard).

3.1.4 Stage 4: Fetch stack

Stack contents are fetched in this stage, unless the reference is supposed to be modified by an instruction present in the following stages, in which case this stage stalls. Operations and fetched values are passed to the *Execute* stage, while data moving instructions bypass *Execute*, if idle, and go directly into *wbfifo*. In order to accelerate the pipeline execution, if the required references are about to be modified by the *Write-back* stage, the operands are forwarded directly from the *wbfifo*.

3.1.5 Stage 5: Execute

The *Execute* stage dequeues two values and an operation identifier from the *fofifo*, executes the operation to obtain a result. Conditional branches are partially handled here, as the operation is simply discarded if the condition is false or passed on to the next stage if the condition is true. Thus, along with the context received from *exfifo*, a destination and a single value are enqueued further in the *wbfifo*.

3.1.6 Stage 6: Write-back

Finally, the values dequeued from *wbfifo* are written back to the right destination (register or stack address) in the *Write-back* stage. Not all registers are available for read or write. For example, OPD can only be read while MWA and MRA may only be written. However the access control is managed within the *Registers* module. Furthermore, this is the stage that may issue a pipeline flush and a context roll-back in the case when PC and JPC are explicitly changed. In addition, this stage also controls the bytecode cache, by issuing a cache refill when explicitly requested by the micro-code (through the CACHECTL register). Having explicit cache refills on method invokes and returns, makes the timing analysis easier. This, in turn, contributes to the predictability of the processor, required in real-time embedded systems.

3.2 Programming Aspects

Embedded Java environments are usually resource and time constrained systems, unlike desktop environments. Many embedded Java solution implement only a limited number of bytecodes and are based on a reduced class library, sufficient for the target application. BLUEJAMM specific solutions are described in the following.

3.2.1 Micro-code Generation

Given the varying complexity of the Java bytecodes, it makes sense to use a micro-programmed processor architecture instead of trying to implement every bytecode as a separate instruction. Simpler bytecodes (i.e. operations, loading constants) would be implemented as one or a few micro-instruction sequence, while the more complex ones (i.e. invokes, new) would require longer micro-programs. Depending on the micro-code, that is, what sequence of micro-instructions needs to be executed for each bytecode, there are four tables/ROMs that need to be generated.

The *micro-ROM* contains the micro-instruction sequences for all the implemented bytecodes. The *BC2microA* table maps bytecodes to micro-code addresses. The *jump table* translates each of the available indexes (up to 32) into an address offset used in the micro-code jumps for updating the PC. The initial *stack* containing constants used in the micro-code.

Our solution takes advantage of the capability of the BSV compiler to automatically generate bit encodings for user defined types. As depicted in Figure 3, our method uses a generator, *bluejasm*, which translates the assembler code into an intermediate BSV file, *generator.bsv*. This file, along with the micro-instruction set definition from *types.bsv* is compiled as a stand-alone simulator by the BSV compiler. Finally, this executable (*genrom*), will output the *.hex* memory image files for the aforementioned tables. The advantage is that if the micro-instruction set encoding changes, the *bluejasm* does not require any updates, since the contents of *generator.bsv* file are encoding independent.

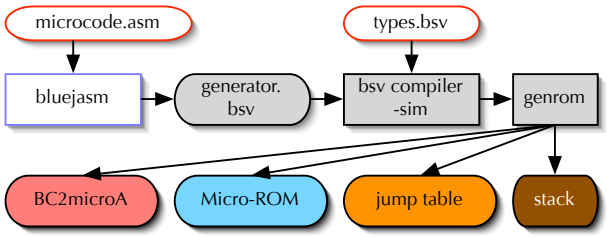

Figure 3. Micro-ROM generation and related tool flow. Colored boxes complete Figure 2.

3.2.2 Run-time environment

As with most embedded Java systems, the runtime class library for BLUEJAMM is smaller than in a desktop environment. The processor, although truly executing bytecodes, uses a specific memory image which has been obtained offline, through a custom class loader and linker *BlueJim*.

The *BlueJim* class loader and linker is an in-house Java written application and it uses the Byte Code Engineering Library (BCEL) to parse *.class* files and generate a proper BLUEJEP executable. The main task of *BlueJim* is to transform all generic references into memory addresses associated with the classes and constants used in the application. Furthermore, all the unused methods and classes are discarded, virtual tables re-organized if necessary, in order to minimize the image size. Finally, the image generator also translates native calls (INVOKESTATICs of methods from the *Native* class) into custom bytecodes, which are not used in the standard JVMs. These custom bytecodes correspond to BLUEJEP specific operations, such as direct memory access, internal register and stack access and some memory management functions.

3.3 Extensions

The architecture described above can be easily extended with new operations and registers. For example, adding a multiplication operation would require only adding another element to the *Operations* enumeration and handling that case in the *Execute* stage, which means one line of code. Multi-cycle operations can also be added, but are a bit more complex and require introducing some control registers. For example, the cache fill is a multi-cycle operation controlled through the CACHECTL register. As another example, the MMU operations are carried out through writing and reading specialized control registers, as detailed in Section 4.

Adding new registers is easy, as well. It requires adding new elements in the *Registers* enumeration type and handling reading and writing to these new registers in the module gathering the register access. These new registers may be real registers, that hold information, or virtual ones that only start or complete a specific action (CACHECTL). New data hazards that may have appeared due to the new register functionality may be handled by extending the stall function found in the *Decode* and fetch registers. For example, a read from an MMU register stalls if an MMU operation is about to start (write to the MMU control register).

4 Memory Management

Every respectable Java environment must have an automatic memory management, or one of the advantages of using Java is lost. BLUEJAMM offers two configurations for automatic memory management with garbage collection (GC) - one is the pure software approach, and the other is by using the specialized hardware memory management unit (see section 2). Both use the same data structures for object instances, handles, class and garbage collection information, as depicted in Figure 4. Furthermore, both configurations use the same GC algorithm, of mark-compact kind. Briefly, a mark-compact garbage collection occurs in two phases (refer to [8] for more details):

Mark Live objects are marked by checking all the references reachable from the stack variables.

Compact The heap is compacted by moving marked objects towards lower addresses.

As the objects are moving around during the compacting phase of the GC cycle, the physical address of the objects changes. This led us to using unique and life-long object identifiers (handles) instead of object pointers in our system. Although this adds another level of indirection to the object access, it makes memory management easier. This is all transparent to the programmer, the handles being administered by the memory manager. These handles are located in

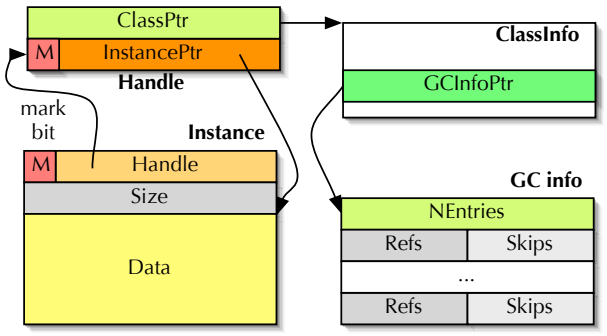

Figure 4. Handles, Classes, and Objects data structures in BluEJAMM

a special memory area, at high addresses, that grows automatically if more handles required (see Figure 5). Additionally, using this specialized area makes identifying handles easier during the marking phase. At the beginning of this phase, the stack must be scanned for all handles, which are the roots for all reachable objects in the heap. A purely conservative approach could consider all stack words as handles, while an exact approach would make sure that each stack value is an actual reference. BLUEJAMM adopts a midway strategy, by considering all stack words within the handle area to be valid root handles, although some of these might be values rather than references, leading occasionally to floating garbage (no impact on functionality).

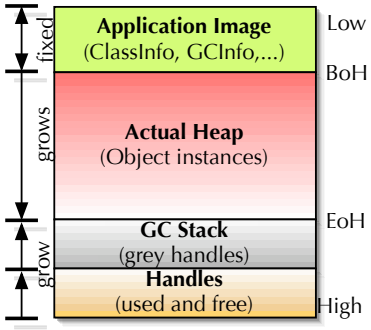

Figure 5. Memory organization in BluEJAMM

The software memory management employs a stop-the-world (STW) garbage collection strategy, which means that during a GC cycle no other operations are performed. It is also allocation driven, meaning that a GC is forced whenever there is not enough free heap space for a new object. Currently, the hardware implementation is also a STW and allocation driven, but extensions towards fully concurrent GC are possible and under way (see [8]).

The hardware MMU control and access in BLUEJAMM is carried out through four specific data registers (MMCtl1–4) and one command register (MMCmd) introduced in the BLUEJEP registers with the operations given in Table 2.

Table 2. MMU registers in BlueJEP

Register	Read	Write
MMCtl1	free heap base	heap base (*init* in)
MMCtl2	free heap end	heap end (*init* in)
MMCtl3	handle (*new* out)	object size (*new* in)
MMCtl4	N/A	object type (*new* in)
MMCmd	N/A	*new* or *forceGC*

As the BLUEJEP processor, the MMU was implemented in Bluespec System Verilog and takes advantage of the finite state machine (FSM) modeling support found in there (*StmtFSM*). An example of how easy is to implement an FSM in BSV is given in Listing 1. In more detail, the MMU is a hierarchy of several FSMs, each implementing a specific task as detailed in Figure 6. The *rootHandles* FSM goes through the stack, marks and pushes possible handles into the grey handles stack. The *Mark* FSM pops handles from the grey handles stack and uses the *GCInfo* to detect more references, that are pushed back in to the grey handles stack. The *Compact* FSM sweeps the heap and moves marked objects in a contiguous space. More handles can be created if necessary through the *extendHandles* FSM. Finally, the *newObjNoCheck* initializes the object handle and contents with necessary pointers. Note that the MMU is highly integrated with the processor, as it needs access to the stack and the SP during the root handle marking step.

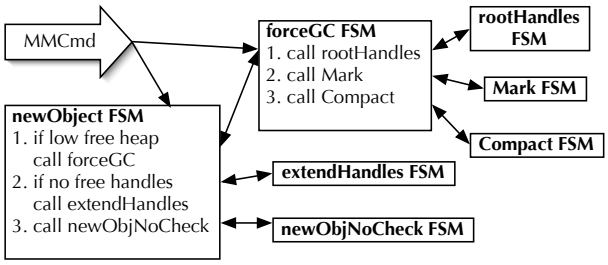

Figure 6. The FSM hierarchy of the MMU

4.1 Real-Time Aspects

For embedded systems that are not also real-time systems, the current implementation of memory management in BLUEJAMM is sufficient. One of the problems that render typical Java environments improper for real-time systems is the timing of memory management functions, in particular garbage collection. The exact duration of a *Stop-the-world* garbage collection is highly dependent on the object configuration, making it hard to predict accurately and with a worst case execution time that is hardly useful. Nevertheless, solutions for real-time garbage collection do exist ([7, 8, 17]), and they can be implemented on BLUEJAMM. In particular, using a concurrent hardware MMU seems to be the choice giving the highest performance. We are currently extending the MMU to support concurrent garbage collection, through a locking mechanism as in [8] and bus snooping designed to guarantee progress.

Listing 1. The *rootHandles* BSV description

```
1 Reg#(Byte) baseP <- mkConfigReg('INITIAL_SP);
2 Reg#(Byte) stackP <- mkConfigReg('INITIAL_SP);
3 Reg#(Word) maybeH = temp; // a shared register

5 function Bool isNotHandle(Word w);
6     return (pack(w)[0] == 0) ||
7         (w < zeroExtend(freeHeapEnd)) ||
8         (w > zeroExtend(heapEnd));
9 endfunction

11 Stmt roothandles_stmt = (seq
12             while(baseP != stackP) seq
13                 action
14                     maybeH <= thestack.sub(stackP);
15                     stackP <= stackP - 1;
16                 endaction
17                 if(!isNotHandle(maybeH))
18                         heystack.push(truncate(maybeH));
19             endseq
20         endseq);

22 FSM roothandles_fsm <- mkFSMWithPred(roothandles_stmt,
23         phase == RootHandles);
```

Note also that virtual memory, as implemented in other processors, is neither required nor suited for our BLUEJAMM environment for several reasons. First, the memory protection is handled through the Java mechanism of accessing objects only and bounds checking for arrays. Second, the memory fragmentation is taken care of the compacting phase of the garbage collection cycle. Third, real-time predictability would suffer dramatically with the implementation of a virtual memory mechanism.

5 Implementation and Results

The whole process of developing the BLUEJAMM system was rather fast, leading to a flexible and modular system due to the BSV based design flow. In particular, the BLUEJEP processor took about four months to develop, including various version, which is half the time needed for a similar configuration, based on JOP, in VHDL. The fact that we had almost no knowledge of Bluespec SystemVerilog before starting the development is also worth noticing. The MMU with garbage collection in BLUEJAMM took only one week to write and test due to the FSM support

in BSV. Granted we did have experience from a previous VHDL solution (see [8]), the design time is still impressive.

The number of code lines is another indicator of the BSV based design flow efficiency. Our BLUEJEP processor is described on about 1300 BSV code lines (2000 Verilog lines after compilation), while a similar VHDL design based on JOP (shorter pipeline) takes around 1900 lines. Our MMU written in BSV takes about 600 code lines, while the a VHDL written similar version takes 1700 lines. During development, more lines were added for testing, debugging and exploration purposes, not considered above.

The target platform used to implement our hardware architecture is an evaluation board based on a Xilinx Virtex-II (XC2V1000, fg456-4) FPGA. All the results presented in this paper were obtained using the Xilinx ISE 9.1i tool chain for the VHDL flow [23]. In addition, we used the 2006.11 version of the Bluespec SystemVerilog compiler in the BSV design flow. The designs were incorporated in systems built with Xilinx EDK 9.1. Once the system was running in hardware, we used Xilinx ChipScope [22] to tap various signals, including the bus and compare the values obtained from the real hardware against values obtained using the Bluespec standalone simulation of our design. This step helped to detect and fix bugs, and to confirm that the implementation behaves similarly to the high-level specification.

When it comes to device area, using a high-level of abstraction language, such as BSV, it is rather expected that the synthesized designs would be rather large. In fact, the BLUEJEP processor takes double the area of the aforementioned JOP VHDL version (3460 vs. 1723 slices). The logic occupies however almost the same number of slices, the difference coming from the memory elements. With the hardware MMU, the area of BLUEJEP increases to 4340 slices. The performance however seems to be within acceptable range. The synthesis tool reports a maximum clock speed of 85 MHz for BLUEJEP alone, and 64 MHz with MMU. Specific efforts for decreasing the critical path with the MMU were not made up to this point, but we are positive that the performance can be increased over these figures.

Performance is, however, not given just by the clock frequency, but also how fast bytecodes (sequences of micro-instructions) can be executed. These figures are dependent on the memory access speed, caching strategies, and micro-program. For similar configuration (2 clock cycle access memory), benchmarking reveals that BLUEJEP performance is not far from JOP [15], as detailed in Table 3.

As for the performance gain by using the hardware MMU compared to the software solution, Table 4 shows profiling data for a simple list handling application. Note that the hardware GC is around twenty times faster than the software solution. The number of clock cycles per executed bytecode increases from 6 to 7 because of the stop-the-world nature of the GC carried out by the MMU now.

Table 3. Execution time in clock cycles for BLUEJEP and JOP for several bytecodes.

Bytecode(s)	JOP	BlueJEP
iload iadd	2	3
iinc	11	13
ldc	9	12
if_icmplt taken	6	23
if_icmplt n/taken	6	8
getfield	23	38
getstatic	15	18
iaload	29	45
invoke	126	166
invoke static	100	111

Table 4. The profile of a simple application on BLUEJAMM with software GC and with MMU

Profile	SoftGC	MMU	MMU/SoftGC
used bytecodes	24810	10304	42%
clock cycles/byte	6	7	117 %
cache fills	1601	675	42%
mem accesses	9063	3139	34%
GC used clocks	49214	2626	5%
total clock cycles	168977	73981	44%

6 Related Work

Besides JOP [16], which was the starting point for our processor, several designs for Java embedded processors were reported in the research community. Some of these are available as soft-cores or even chips, many designed for embedded systems and few even for real-time applications. Memory management is often a very simple, software implementation at best. Relevant approaches are briefly listed here. A detailed comparison between these processors can be found in [15].

Sun's PicoJava-II [18], freely available, is arguably the most complex Java processor currently, a re-design of an older solution which was never released. Its architecture features a stack-based six stages pipelined CISC processor, implementing 341 different instructions. Folding of up to four instructions is also implemented.

aJile's JEMCore, based on Rockwell-Collins' JEM2 design, is available both as IP or standalone processor known as aJ-100 [1], a 0.25μ ASIC operating at 100MHz. The 32-bit core is a micro-programmed solution, comprising ROM

and RAM control stores, an ALU, barrel shifter, and a 24-element register file. JEMCore implements, besides native JVM bytecodes, extended bytecodes for I/O and threading support, along with floating point arithmetic.

DCT's Lightfoot 32-bit core [6] is a hybrid 8-bit instruction, 32-bit data path Harvard dual-stack RISC architecture. The core comprises a three stages pipeline, with an integer ALU including a barrel shifter and a multiplication unit. Lightfoot has 128 fixed instructions and 128 reconfigurable, soft-bytecodes.

Vulcan Machines' [20] Moon2 is a 32-bit processor core available as a soft IP for FPGA or ASIC implementation. The Moon core features an ALU, a 256-element internal stack, optional code cache, and a micro-program memory for the operation sequence required by each bytecode.

The Komodo micro-controller [12] includes a multithreaded Java processor core, which is a micro-programmed, four stages pipeline. Its remarkable feature is the four-way instruction fetch unit, with independent program counters and flags, for hardware real-time scheduling of four threads.

FemtoJava is research project focused on developing low-power Java processors for embedded applications. One of the versions features a five stages pipelined stack machine later extended to a VLIW machine [2], synthesized for an FPGA. Data about the FPGA make, clock speed, and whether it actually ran on the FPGA are unclear.

7 Conclusion

This paper described BLUEJAMM, a Bluespec embedded Java architecture with memory management. The core of the architecture is the BLUEJEP processor and its special memory management unit, both developed in Bluespec SystemVerilog. The processor is a native Java micro-programmed core, implemented as a six stage pipeline stack machine. The MMU offers a hardware alternative to software memory management and garbage collection, for solutions willing to trade device area for performance. The architecture described herein is flexible, modular, specified at a high-level of abstraction and offers a reasonable performance for prototyping embedded Java systems. BLUEJAMM has been implemented and tested on a Xilinx Virtex-II FPGA in a relatively short time, thanks to the support offered by the BSV-based design flow.

References

[1] AJile Systems. http://www.ajile.com.

[2] A. C. S. Beck and L. Carro. A VLIW low power Java processor for embedded applications. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 157–162, New York, NY, USA, 2004.

[3] Bluespec, Inc. http://www.bluespec.com, 2007.

[4] N. Dave. Designing a processor in Bluespec. Master's thesis, MIT, Cambridge, MA, January 2005.

[5] N. Dave, M. Pellauer, S. Gerding, and Arvind. 802.11a transmitter: A case study in microarchitectural exploration. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE'06)*, pages 59–68, July 2006.

[6] Digital Communication Technologies. Lightfoot 32-bit Java processor core. data sheet, September 2001.

[7] S. Gestegard-Robertz and R. Henriksson. Time-triggered garbage collection. In *Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems*, June 2003.

[8] F. Gruian and Z. Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Asia-Pacific Computer Systems Architecture Conference*, pages 281–294, 2005.

[9] F. Gruian and M. Westmijze. BlueJEP: A flexible and high-performance Java embedded processor. In *The 5th Int'l Workshop on Java Technologies for Real-time and Embedded Systems, JTRES'07*, September 26–29 2007. to be presented.

[10] J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI '99: Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration*, pages 595–619, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[11] A. Ive. Towards an embedded real-time Java virtual machine. Lic.Thesis 20, Dept. of Computer Science, Lund University, June 2003.

[12] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, February 2003.

[13] A. Nilsson. Compiling Java for real-time systems. Licentiate thesis, Lund Institute of Technology, 2004.

[14] RTJ Computing. Simple real-time-java, http://www.rtjcom.com/, July 2007.

[15] M. Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[16] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, January 2005.

[17] W. Srisa-an, C.-T. D. Lo, and J. M. Chang. Active memory processor: A hardware garbage collector for real-time java embedded devices. *IEEE Transactions on Mobile Computing*, 2(2):89–101, April–June 2003.

[18] Sun. PicoJava-II microarchitecture guide. Technical Report 960-1160-11, Sun Microsystems, 1999.

[19] Sun Microsystems, Inc. *J2ME Building Blocks for Mobile Devices*, May 2000.

[20] Vulcan Machines Ltd. http://www.vulcanmachines.co.uk/, August 2007.

[21] R. E. Wunderlich and J. C. Hoe. In-system FPGA prototyping of an Itanium microarchitecture. In *International Conference on Computer Design*, October 2004.

[22] Xilinx. *ChipScope Pro Software and Cores User Guide*, v9.1.01 edition, January 2007.

[23] Xilinx Inc. http://www.xilinx.com/, 2007.