Embedded Systems, Advanced Course
ZUMA Arcade Game
Final Report

Ang, Lay Hong (sx07la2@student.lth.se)
Chaiwat Sittisombut (sx07cs2@student.lth.se)
Lim, Wee Guan (sx07wl2@student.lth.se)

November 5, 2008

Abstract

This report documents the development of an embedded version of an arcade game that runs on a Xilinx Spartan3E-1200 FPGA on a Digilent Inc. Nexsys2 development board. A Microblaze processor, together with Xilinx provided hardware blocks and custom VHDL blocks were coupled together with the game software to allow the User to play the game via a PS/2 keyboard and to receive outputs from the game via a VGA monitor and the seven-segment display on the board. The resulting system consumes 22 kB of the 32 kB of memory available to the Microblaze for code and data storage and utilizes 95 % of the available slices on the FPGA.

Contents

Chapter 1

Introduction

Ang Lay Hong, Chaiwat Sittisombut, Lim Wee Guan

1.1 ZUMA

The objective of the game developed is to eliminate all the colored balls rolling on the VGA screen along a specific path, before the sequence of colored balls reach the exit. To prevent the colored balls from reaching the exit, the player can eliminate the balls by firing a colored shooter ball from the stand at the bottom of the screen. When three or more balls of the same color come in contact during the firing, the balls explode and points are awarded. The collision will result in collapsing of the neighbouring balls, that may trigger more chain reactions. If three or more balls of the same color come in contact due to the collapsing, they are destroyed and more points are awarded.

If there are no two or more balls of the same color being hit by the shooter ball, the shooter ball is inserted into the sequence of balls. The shooter ball is allowed to move horizontally across screen to aim at the sequence of balls.

There are 4 levels of difficulty implemented for the game. The number of different colors for the sequence balls increases as level of difficulty increases. A game level is completed when all the balls on the screen are destroyed. The game is over when the first ball in the sequence hits the exit. A screenshot of the game described is shown in Figure 1.1.

1.2 Obtaining and Building the Project

The source and project files can be obtained from the EDA385 Course website at `http://www.cs.lth.se/EDA385/HT08/`.

The project archive contains a Xilinx EDK project for the Nexys2 Development Board. The full hardware requirements for running this project is given in Section 1.3. The project file requires Xilinx Platform Studio version 10.1 or later to build.

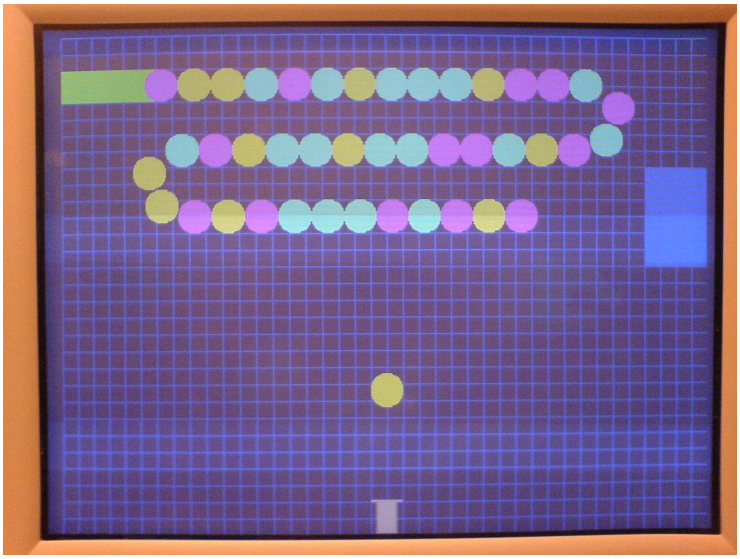


Figure 1.1: Screen shot of the simplified Zuma game on Nexsys-2 board

1.3 Hardware Requirements

This project was developed and tested using Digilent Inc. Nexys2 Board with a SPARTAN3E-1200 FPGA. It will probably run on any other development board with a comparable or larger FPGA in addition to the following hardware peripherals.

- VGA Port with support for at least 8-bit colour
- PS/2 port capable of sourcing 5 V
- 4 digit 7-segment display

Additional hardware requirements include a PS/2 keyboard to control the game, and a VGA monitor capable of supporting 640×480 display resolution to act as a display.

1.4 Organization of the Report

This report documents the design of the ZUMA Arcade game running on a Xilinx FPGA development board.

Chapter 2 describes the overall architecture of the project, and describes the hardware and software architecture. The hardware architecture briefly describes the hardware components in the project and the software architecture describes briefly the software that is used for configuring the hardware peripherals.

Chapter 3 describes in detail the development of the custom hardware blocks, namely the Seven Segment Display and the VGA Controller.

Chapter 4 describes in detail the development of the game software and algorithms that are running on the Microblaze platform.

Chapter 5 describes the tasks undertaken to perform integration of the hardware and software, as well as the testing that was done to ensure correctness of operation.

Chapter 6 concludes the report and talks about the lessons learnt and interesting insights obtained from the work done on the project.

1.5 Contributions

The following section documents the contributions of each individual team member to the project.

Task	Contributor
Architecture Design	
Hardware Architecture	Lim, Wee Guan
Software Architecture	Ang, Lay Hong
Hardware Development	
VGA Controller	Lim, Wee Guan
Software Control Modules	
Main Controller	Ang, Lay Hong
Hardware Interfacing	Chaiwat Sittisombut
Game Algorithm Modules	
Moving & Shooter balls management	Ang, Lay Hong
Collision Detection & Handling	Chaiwat Sittisombut
Integration & Testing	Lim, Wee Guan Ang, Lay Hong Chaiwat Sittisombut

Chapter 2

Architecture

Ang Lay Hong, Lim Wee Guan

This chapter describes the overall architecture of the Project. Section 2.1 describes the hardware architecture and all the hardware peripherals that are connected to the Microblaze and Section 2.2 describes the software aspects of controlling the various peripherals.

2.1 Hardware Architecture

Lim Wee Guan

The hardware architecture selected is based on the PLB Bus provided by Xilinx. In this architecture, the Microblaze processor sits on the PLB bus, and is the only Master on the Bus. The remaining blocks, namely the VGA Controller, UART, PS/2 Controller, Seven Segment Controller and Interrupt Controller are attached as slaves on the PLB Bus. The block diagram of the various blocks connected to the PLB is shown in Figure 2.1

The various blocks in Figure 2.1 are described briefly in Sections 2.1.1 to 2.1.5.

2.1.1 PS/2 Controller

The PS/2 core interacts with the PS/2 keyboard, reads in the scan codes sent by the keyboard and interrupts the Microblaze when data is received. The data received by the PS/2 Controller is written into a memory mapped data register. This IP core is provided by Xilinx and is used unmodified in this project.

The main problem faced in getting the keyboard to be recognized by the Nexys2 board although it was shown earlier via the Digilent BIST code that the mouse was working correctly. It was found later that the keyboards used expected 5 V input and the jumper on the Nexys2 board was set incorrectly to send out 3.3 V. After changing the jumper settings, the keyboard worked correctly.

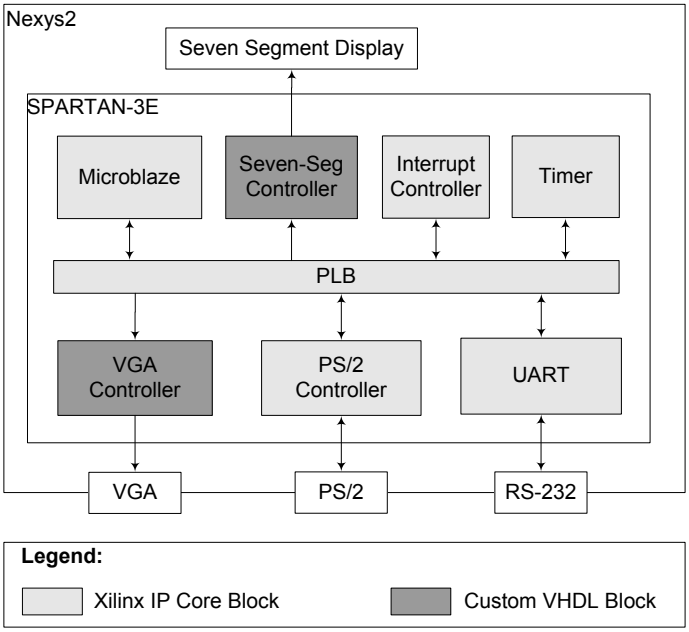


Figure 2.1: Block Diagram of Hardware Components on PLB Bus

2.1.2 Seven Segment Controller

The seven segment display interfaces between the Microblaze and the seven segment display. The eight segments of each number (7-segments per digit and decimal point) have cathodes that are common across all four numbers and a single anode for each of the numbers to reduce the number of FPGA pins used on the FPGA. Therefore, it is necessary to perform a scanning pattern to light up each of the numbers alternately at a sufficiently high rate so that persistence of vision will make it appear to the viewer as if all four numbers are lighted up at the same time.

The Microblaze writes the four digits to be displayed on the seven segment display by writing a 32-bit word to a memory mapped address. Each of the 8-bits in this 32-bit word will represent a single digit to be displayed.

The design of the Seven Segment Controller is covered in detail in Section 3.1.

2.1.3 VGA Controller

The Nexys2 board supports 8-bit color output via a DB-15 connector. Eight pins of the FPGA are connected to voltage divider circuits that together with the 75 Ω termination resistance of the VGA monitor allows for display of up to 256 different colours. Two other signals, namely the `HSYNC` and `VSYNC` are also sent by the FPGA to provide the sync pulses that are necessary to set the frequency of current in the deflection coils.

A VGA Controller is needed to produce the synchronizing signals as well as the VGA data to be displayed. The VGA Controller will also need to interface with the PLB Bus to allow the Microblaze to control the images that are being

displayed. Data from the Microblaze is passed to the VGA Controller using 50 memory mapped registers, each representing the coordinate of a single object that is to be displayed on screen. The VGA Controller will then paint the respective objects onto the screen.

The design of the VGA Controller is covered in detail in Section 3.2.

2.1.4 Timer

The Timer is an IP core provided by Xilinx as part of the EDK suite. It provides up to two timer counters and will trigger an interrupt when it has reached the desired count. Configuration of the Timer is done via memory mapped control registers.

2.1.5 UART and Interrupt Controller

The UART and Interrupt Controller modules are IP cores provided by Xilinx as part of the EDK suite. The UART is used to provide an RS-232 interface to the Microblaze which is then used as a debug console during development.

The Interrupt Controller receives interrupts from the Timer and PS/2 cores, and consolidates them into a single interrupt line to the Microblaze. The Microblaze then reads the memory mapped registers for the Interrupt Controller to check for the source of the interrupt before running the requisite interrupt service routine.

2.1.6 Deviations from Proposed Design

The proposal outlined a custom PS/2 controller that triggered interrupts only when keys related to the game are depressed. The rationale for this was to minimize the interrupt load on the Microblaze in event of random key-presses. This was not implemented because of time constraints as it would have been necessary to design, implement, test and more importantly, integrate the new PS/2 controller into the system. Moreover, it was felt that it would have been more productive to optimize the software to handle the load rather than to modify the hardware.

The seven segment display was originally envisioned to be driven using the generic GPIO core provided by Xilinx. This approach was not taken as it was felt that the Microblaze is not suitable for performing the regular refresh that the seven segment display requires. On the other hand, this task is trivial for hardware that is driven by a clock signal. Therefore, it was decided that a custom IP core will need to be developed to drive the seven segment display.

2.2 Software Architecture

Ang Lay Hong

The following section describes the software architecture for the Zuma game platform. An overview of the software framework is illustrated in Figure 2.2

The software framework is based on Microblaze, together with several IP cores such as the PS/2 controller, a timer, the interrupt controller and block

ram. PS/2 controller is used to receive keyboard events from a PS/2 keyboard for controlling of movement of shooter ball. The timer is made up of two timer counters which are used for controlling motions of moving objects. The interrupt controller is required for receiving interrupt-driven timer and PS/2 events. This is more efficient as compared to polling-based timer counter and PS/2 events, which will waste unnecessary CPU cycles and limit the amount of computations that can be performed. Block ram is required to store the code and data of microblaze. As the size of block ram in SPARTAN3E-1200 family is 504 kb (i.e. 63 kB), the maximum usable size of block ram in Microblaze is limited to 32 kB.

The software framework accesses the VGA controller core and 7-segment display core via memory-mapped addressing.

2.2.1 Timer Counter

As objects on VGA screen need to be updated once every defined amount of time, a timer counter may be used as described as follows.

The timer counter is configured to countdown from a specific reset count number. For instance, given a defined reset value of 5000, the timer counter is configured to start decrementing from 5000 until the value 0 is hit. The reset value as be seen as the period of the timer counter. The timer counter interrupt signal is bound to the interrupt controller such that once the timer counter value hits 0, the timer interrupt signal is received by the interrupt controller. The interrupt controller sends the timer interrupt signal to Microblaze and the interrupt service routine (ISR) is called to set a flag. The main routine upon detection of the flag, will perform the necessary update, while the timer counter wraps around and continue to count down from counter value 5000. Since the processor is running at a clock rate of 50 MHz, a desirable delay of say 100 ms requires a counter reset value of $5,000,000$. Figure 2.3 illustrates the concept described.

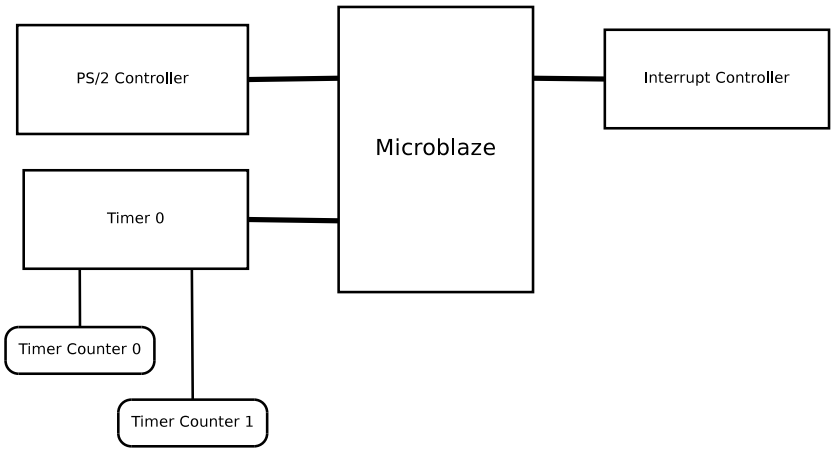


Figure 2.2: Block diagram of Software Framework

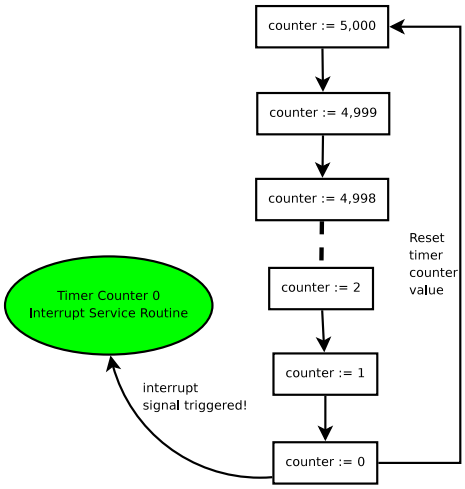


Figure 2.3: Concept of interrupt-driven timer counter event

2.2.2 PS/2 Controller

XPS has a PS/2 Controller core that is used in this project. The PS/2 controller detects keyboard events and stores scan codes received in registers. Microblaze can receive PS/2 events either by blocking/non-blocking polling mode or interrupt-based mode.

The PS/2 controller may be configured to be triggered via an interrupt signal `PS2INT`, that is bind to the interrupt controller. PS/2 data bytes are received via the PS/2 clock pinout `PS2_CLK` and data pinout `PS2_DATA IO`.

To receive PS/2 events in interrupt mode, a non-blocking receiving function such as `PS2_ReceiveByte()` is first called. This enables the PS/2 interrupt signal. When requested number of bytes are received, the PS/2 interrupt signal is sent to the interrupt controller, which in turn interrupts Microblaze. The PS/2 ISR is then called to perform the necessary handling for the received keyboard events.

Chapter 3

Hardware

Lim Wee Guan

This chapter describes custom VHDL blocks that were developed for the project. Section 3.1 describes the Seven-Segment Display controller, and Section 3.2 describes the VGA Controller.

The total FPGA utilization for the project is tabulated in Table 3.1. As observed, utilization of the FPGA is rather high with 95 % of the slices occupied. Observing that 9.37 % of the used 4-Input LUTs are for pass-through indicates that there is some part of the circuit that has many wires congregating at the same circuit, and the FPGA is using some of the 4-LUT as wires to supplement the insufficient wiring resources. The circuit utilizing the large amount of wiring resources is in the VGA Controller and although steps have been taken to reduce the problem, it has not been totally resolved.

Table 3.2 summarizes the resource utilization for each of the two custom VHDL blocks created for this project. The VGA Controller takes up a large proportion of the FPGA and is a prime target for further fine-tuning and optimization in the future. Details of some of the optimizations in the VGA Controller that was undertaken as part of the project is detailed in Section 3.2.8.

Table 3.1: FPGA Utilization

Resource	Utilization	% Utilization
Slice Registers	6,083 of 17,344	35 %
Occupied Slices	8,251 of 8,672	95 %
4-Input LUT Utilization	14,619 of 17,344	84 %
4-Input LUT as logic	12,851 of 14,619	87.9 %
4-Input LUT as route-thru	1,370 of 14,619	9.37 %
Bonded IOBs	55 of 250	22 %
RAMB16s	16 of 28	57 %
BUFGMUXs	2 of 24	8 %
DCMs	1 of 8	12 %
MULT18X18SIOs	3 of 28	10 %

Table 3.2: FPGA Resource Utilization of the Custom VHDL Blocks

Custom Core	Slices	Slice Reg	LUT
VGA Controller	7,347	3,563	10,068
7-Seg Display	68	51	39

3.1 Seven Segment Display

The Seven Segment Display IP core is an independent development. It is integrated into the project for displaying of scores for the Zuma game. *See attached file from the developer.*

3.2 VGA Controller

Lim Wee Guan

The VGA Controller serves an the interface with the Microblaze and is responsible for the generation of the necessary synchronization and data signals to be sent to the VGA monitor.

Section 3.2.1 details the development approach taken to design and test the VGA Controller as a standalone module.

3.2.1 Development Approach

The VGA Controller is a relatively large hardware module written in VHDL. As such, it is broken down into components and tested in a piece-wise fashion with small component-wise test-benches before being put together in a hierarchical fashion. The test-benches are simulated using ModelSim Xilinx Edition (XE) and checked for correct operation.

Once the entire VGA controller has been assembled and tested to operate correctly in a ModelSim simulation, the VGA controller was incorporated into a wrapper module that has a corresponding User Constraints File (`.ucf`) file. The entire wrapped design was then synthesized using ISE into a standalone `.bit` file that was loaded onto the Nexys2 board. This is to ensure that the VGA Controller is able to operate in a standalone manner. The testing of the VGA Controller as a standalone `.bit` file is important to be performed prior to integration with Microblaze as it will narrow down problems seen after integration to the PLB interface since the VGA Controller has been shown to be able to operate correctly previously.

Interfacing of the VGA Controller module is performed by simply replacing the standalone wrapper with the EDK generated `user_logic.vhd` file, with the VGA Controller's top-level module instantiated within. The details of the hardware integration of the custom written cores to the PLB/Microblaze is covered in Section 5.2

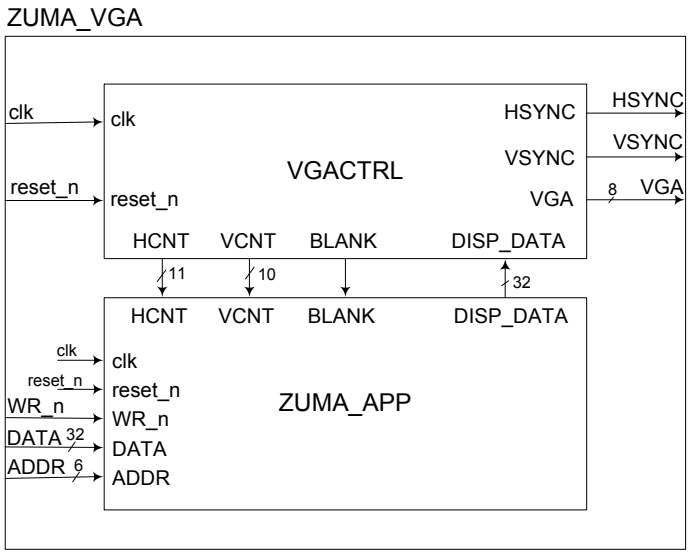


Figure 3.1: Top Level Block Diagram of VGA Controller

3.2.2 VGA Controller Architecture

At the top-most level of the hardware hierarchy, the VGA Controller consists of two main blocks, namely the `VGACounter` and the `ZUMA_APP`. The separation of the VGA Controller into `VGACounter` and `ZUMA_APP` modules is so that it is possible to reuse the `VGACounter` in other applications since it is a generic 640×480 VGA Counter. A block diagram showing the top-level of the VGA Controller is given in Figure 3.1.

The `VGACounter` block performs the counter tasks as well as the data transmission tasks. The counter tasks cover the generation of `HCOUNT`, `VCOUNT`, `blank` position signals, and the `HSYNC` and `VSYNC` synchronization signals. `HCOUNT` and `VCOUNT` track the horizontal and vertical positions of the electron gun, and `blank` indicates whether the electron gun is in the active display area or not. The `HSYNC` and `VSYNC` signals are used to synchronize the VGA Monitor. Finally, the `VGACounter` receives the data to be transmitted from the `ZUMA_APP` module and transmits the data in time with the `HSYNC` and `VSYNC` signals. The details of the `VGACounter` are presented in Section 3.2.3

The `ZUMA_APP` is the application specific part of the VGA Controller that is specific to the ZUMA application. It is responsible for the generation of the data to be transmitted based on color and coordinate information that is sent by the Microblaze. It should be noted that although the colour depth of the Nexys2 board is only 8 bits, data is transferred in blocks of 32 bits, which corresponds to 4 pixels. This is because all image data is processed in blocks of 4 pixels. The rationale for using 4 pixels/block is described in detail in Section 3.2.4.

3.2.3 VGACounter

The `VGACounter` consists of 2 modules, namely `VGACount` and `VGA_Output_IF`. The former is responsible for producing the VGA Control signals and the latter

is responsible for taking in the 32-bit wide (a block of 4-pixels) data and transmitting them out one pixel (byte) at a time. These two modules are described in the paragraphs that follow.

`VGACount` does the mundane task of taking the input 50 MHz clock and using it to generate the `HSYNC`, `VSYNC` for synchronizing the monitor, as well as `HCOUNT`, `VCOUNT` and `blank` signals that are used to track position of the pixels being displayed. The key feature of `VGACount` is that `HCOUNT`/2 corresponds to the x-coordinate of the current pixel and `VCOUNT` corresponds to the y-coordinate of the current pixel which makes it easy for any application using these signals to compute the current pixel location. The reason for the division of 2 in `HCOUNT` is that for a 50 MHz clock, the display time is equivalent to 1280 clocks and there are only 640 horizontal pixels which corresponds to a division by 2 in `HCOUNT` to obtain x-coordinate.

The `VGA_Output_IF` consists of a shift register that shifts out 8-bits (one pixel worth of data) every 2 clock cycles (one pixel duration). At periodic intervals of 8 clock cycles, a new block of 32-bit data is fed into the shift register to be shifted out. It is worthwhile to note that this shift register introduces a one-block delay between the data being produced by `ZUMA_APP` and the time when it is transmitted, which has to be compensated for in `ZUMA_APP`.

3.2.4 ZUMA_APP

The `ZUMA_APP` is the module responsible for generating the data to be displayed on the monitor and a block diagram of this module is shown in Figure 3.2. It consists of two main components, namely `BKGND_GEN` and `FGND_GEN` and a third simpler module that overlays (combines) the foreground pixel data with the background data. The `BKGND_GEN` and `FGND_GEN` modules generate the background and foreground pixel data respectively. As all the data is generated in blocks of 4 pixels, all three modules are triggered to run once every 8 clock cycles. The `BKGND_GEN` and `FGND_GEN` are described in detail in Sections 3.2.5 and 3.2.6.

The overlay module works by simply over-writing the background pixel data whenever there is valid foreground information. The validity of the foreground information is carried in a `pixmap` produced by the `FGND_GEN` module that consists of a `std_logic_vector` of length 4 bits, with each bit position corresponding to a pixel in the data block. A 1-bit in the `pixmap` indicates that there is valid foreground data in that position and thus the background pixel will be over-written by foreground data for that position.

3.2.5 Background Generation

The main function of `BKGND_GEN` is to generate the display data to be sent out to the monitor as a function of the (x, y) coordinates. As this module operates only once every block of 4 pixels, it generates a 32-bit block of data which corresponds to 4 pixels of 8-bit color data.

The x-coordinate is derived from `HCNT` by simply dropping off the least significant bit, which corresponds to `HCNT`/2. With the 50 MHz clock, the display area corresponds to 1280 clock cycles, but there are only 640 display pixels in the x-direction. Hence, a single pixel corresponds to two clock cycles. The y-coordinate is simply the value of `VCNT`, or alternatively `VCNT` with the most

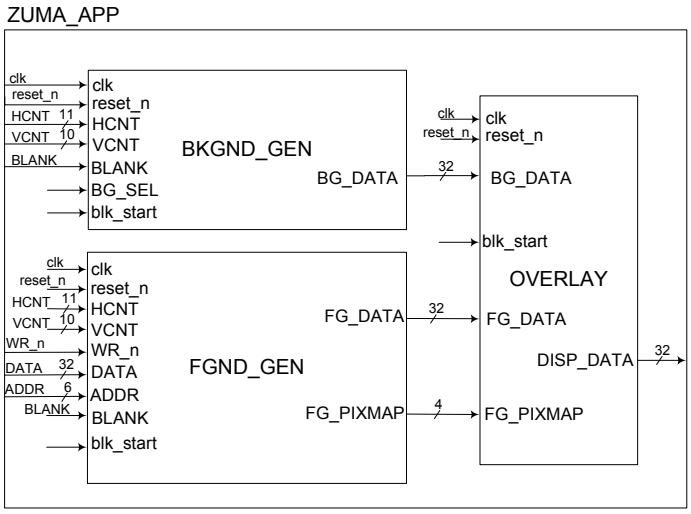


Figure 3.2: Block Diagram for the `ZUMA_APP` Module

significant bit dropped. Deriving the block address within a horizontal row (address of a block of 4-pixels) is simply dropping of the three least significant bits from `HCNT`.

Since all blocks in the `ZUMA_APP` module are triggered to generate one block of display data at the start of each block (`blk_start`), it should be noted that the background data being generated is for the address that is 2 blocks into the future. The reason the delays is because there is a one block delay in the `OVERLAY` module within `ZUMA_APP`, and yet another block delay in the shift register within `VGACounter` that performs the shifting of display data out 8-bits at a time to the VGA display. By generating background data 2 blocks in advance, the display data will be sent out to the monitor at the correct time (in time with `HSYNC` and `VSYNC`) after being delayed through the abovementioned delays.

The objects on-screen that are generated by `BKGND_GEN` include the "graph-paper" like lines in the background, as well as the Entrance and the Exit for the sequence balls.

3.2.6 Foreground Generation

The function of the `FGND_GEN` is similar to that of `BKGND_GEN` except that the output is not only a function of (x, y), but also the data from the Microblaze. The block diagram of `FGND_GEN` is shown in Figure 3.5.

The `CAM` module is a type of Content Addressable Memory that allows for (x, y) coordinates to be fed in, and three sets of outputs stating if one of three active objects in `ObjectDraw` are active and should be triggered to generate foreground data. The `CAM` is described in detail in Section 3.2.7. The `DATA`, `ADDR` and `WR_n` allow for data from the Microblaze to be written into `CAM`.

The `regPosition` module generates the addresses (x and y-coordinates) for the respective modules to operate on. As both `CAM` and `ObjectDraw` take multiple clock cycles to complete their tasks, these two modules are pipelined

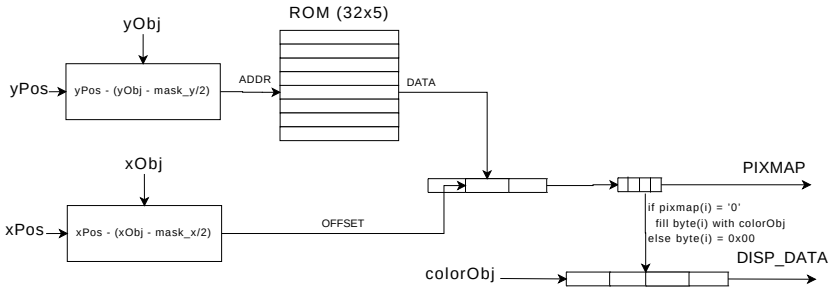


Figure 3.3: Drawing of Object using information from Mask

to operate on separate blocks. When `ObjectDraw` is generating data for block i, `CAM` is working on block $i + 1$. This pipelining is necessary because the total number of clock cycles required by `CAM` and `ObjectDraw` to compute "hit" and to generate the display data exceeds the 8 clock cycles available to process each block. The outputs of `CAM` are held by a pipeline register and is fed to `ObjectDraw` at the start of every block.

In a fashion similar to `BKGND_GEN`, `ObjectDraw` needs to generate foreground data 2 blocks in advance to account for block delays within the design. To support this, `regPosition` feeds addresses (`xPosNext` and `xPosNext`) that are 3 blocks in advance of current `HCNT` and `VCNT` to `CAM`, and addresses (`xPos` and `yPos`) that are 2 blocks in advance of current `HCNT` and `VCNT` to `ObjectDraw`.

ObjectDraw

The `ObjectDraw` provides three draw objects that draw the Shooter stand, and two balls. The objects are drawn on screen using a ROM that contains a mask of the object to be drawn. All objects are drawn within a 32×32 square, and the ROM is 32×5 in size. Each bit in a word in the ROM indicates if a pixel in the particular column should be painted or not. Every address in the ROM corresponds to a row in the square. With this mask, the draw objects can draw arbitrary shapes and fill them with the colour specified in the input data. This concept including the method of calculating address row and the offset is illustrated in Figure 3.3.

The reason why two ball draw objects are needed is because the `ZUMA_APP` operates on a block of 4-pixels at a time. In event that the block being processed falls within two ball masks, e.g. two pixels in the mask of the previous ball object, and the other two pixels in the next ball mask, two ball draw objects will be needed per block. In this "handover" case, two objects will be flagged as active by the `CAM`, and the two ball circuits will be needed to fill up the contents of the entire 4-bit block. This is illustrated in Figure 3.4

3.2.7 Content Addressable Memory

The reason why a Content Addressable Memory like storage system is needed in the `FGND_GEN` is because the coordinates of the ball cannot be sorted within the FPGA, nor is it feasible for it to be sorted by the Microblaze. Any sorting

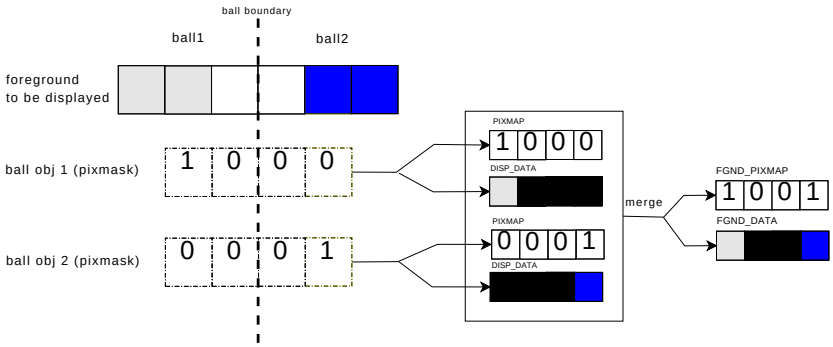


Figure 3.4: Use of Two `DrawObject` to process a single block

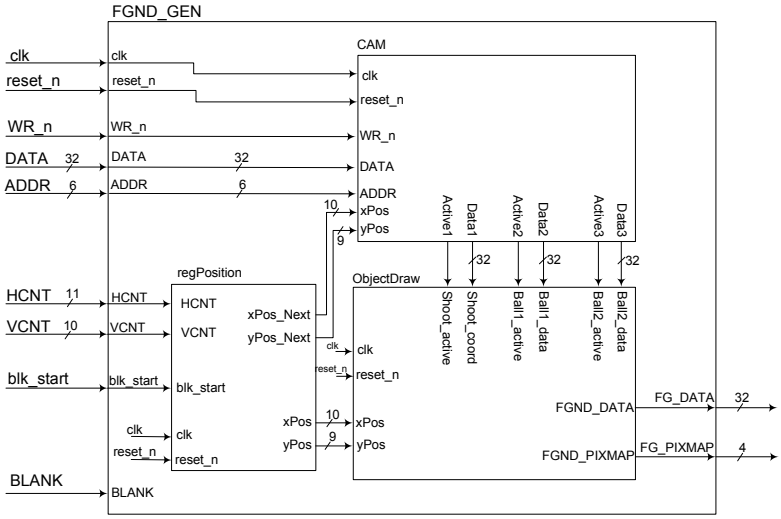


Figure 3.5: Block Diagram for the `FGND_GEN` Module

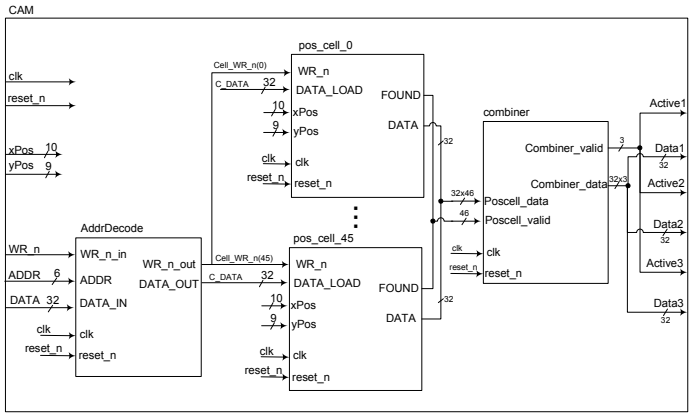


Figure 3.6: Block Diagram for the `CAM` Module

algorithm used must also take into account the scanning of the electron beam, which makes an already non-trivial sort more difficult.

The workaround is to implement memory cells (`pos_cell`) that can be written to in a fashion similar to SRAM using the `WR_n`, `ADDR` and `DATA` signal lines. The search is performed using the (x, y) coordinate signals `xPos` and `yPos`. Each `pos_cell` holds the data pertaining to one foreground object (1×32-bit register). An address decoder module is thus needed to decode the address and to trigger the appropriate `WR_n` line to ensure that data is written into the correct memory cell depending on the address sent by the Microblaze.

Upon receiving a `xPos` and `yPos`, the `pos_cell` computes if the current position coincides the object that it currently holds data for. The computation is similar to that performed in `ObjectDraw` and was described in Section 3.2.6. If a "hit" is found, the data stored within `pos_cell` is put out on the `DATA` port, and `VALID` is set to `'1'`.

The `combiner` module combines all the `VALID` and `DATA` into three DATA and ACTIVE signals. `ACTIVE1` and `DATA1` are special as it is directly wired to `pos_cell0` that contains the data for drawing of the shooter stand. This ensures that the shooter object's data is always passed to the shooter stand draw object, and that it has priority over the ball draw data. The first two `VALID` and `DATA` with `VALID` = '1' for ball draw objects are passed out on `DATA2`, `DATA3`, `ACTIVE1` and `ACTIVE2` to be fed into the `ObjectDraw` module to draw the ball objects. A detailed description of the optimizations made on the `combiner` is described in greater detail in Section 3.2.8.

3.2.8 Problems Encountered with VGA Controller

As outlined at the beginning of the chapter, the VGA controller is the single largest hardware block in the design. The original design for the VGA controller did not have a `CAM`, but instead had an array of ball draw objects (one per object) that was responsible for drawing a single object on screen. However, as each of the ball draw objects takes up rather sizeable resources of the FPGA, the design was not scalable above 20 foreground objects due to lack of FPGA resources. Moreover, the outputs from the ball draw objects consists of 32-bit data and

4-bit pixmap fields, and to combine all these inputs into a single output needs huge numbers of wires congregating in a single point, which used up yet more 4-LUT as routing resources, which in turn used up yet more resources.

A redesign of the VGA Controller led to the current "Time-Multiplexed" design, where two ball draw objects and a single shooter-stand draw object were responsible for drawing all the foreground objects on screen. Although this reduced the resource utilization of using individual draw objects for each foreground object, the wiring problems were merely pushed back into the `CAM` where there are still large numbers of inputs (from `pos_cell` modules) being combined into 3 outputs. The first version of the `combiner` module in the `CAM` was simply a `loop` in a `process` that combined all the outputs from the various `pos_cell` modules. In addition to the large numbers of wires, another problem is the long critical path introduced by this module as the number active objects increased. This design was only scalable up to 32 active objects on screen, and ISE reported the VGA controller's operating frequency as 25 MHz which was unacceptable.

A final redesign was performed, with all efforts focused on `combiner` to reduce resource utilization and to improve critical path. The module was broken down into an array of smaller combiners, each of size 8. By combining them in an array as shown in Figure 3.7, and pipelining them over multiple clock cycles (one clock cycle per stage), the number of wires was reduced, and the critical path shortened significantly with ISE reporting the final design of the VGA Controller as capable of operating at up to 83 MHz. This design is capable of having 46 foreground objects, which was able to meet the project's design goal. It should be noted that although the design of the `combiner` allows for up to 64 active objects, and only 46 objects are used, the unused modules (2 of the `combiner8` in Stage 1) are optimized away during synthesis, and do not occupy any resources on the FPGA.

A further possible improvement that is possible on the VGA Controller is to use locations in the block RAM of the FPGA to store images that can be used for the foreground objects. This will make the images that can be displayed controllable via software and allow for more dynamic objects rather than the current ones that are only filled with one colour. However, the foreground image painting algorithm in this case will also be much more complicated, which given the current FPGA resource limitations may not be possible.

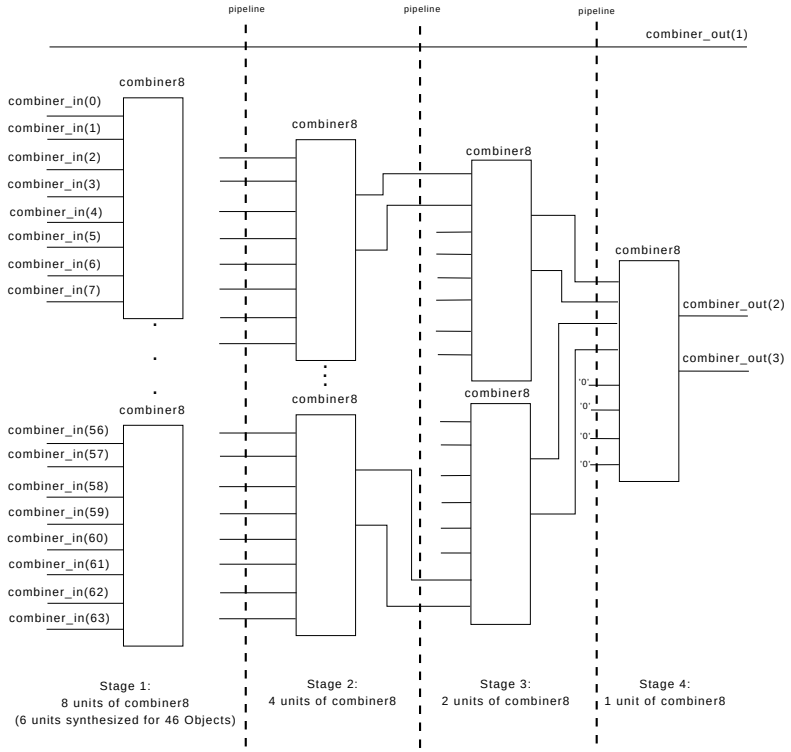


Figure 3.7: Conceptual Block Diagram for the final `combiner` Module

Chapter 4

Software

Ang Lay Hong, Chaiwat Sittisombut

The following chapter describes about the software modules developed in Microblaze. The software development approach is first mentioned, followed by a brief description of the Zuma game algorithm implemented. Implementation work performed on the various software modules are then described. Problems encountered during the midst of the project are mentioned with proposed solutions.

4.1 Software Development Approach

Ang Lay Hong

The software development is carried out concurrently with the hardware developement of VGA and 7-segment controllers. This implies that hardware components are not readily available for testing of the software core during the development time. A more efficient methodology is required to facilitate the development of the software architecture.

The approach adopted is to develop the game algorithm on a `LINUX` host as a simulator whilst hardware development is in progress. Simple on-screen print messages are used to simulate the display of moving balls on VGA screen, with respect to the center pixel locations of the balls.

An obvious advantage of this approach is the readily available debugging toolchain `gdb` to perform more efficient debugging. The debugging of software is much more easier with all debugging information available as needed, as compared to developing the software directly on the target host.

Direct implementation of software on the target poses a great challenge in the debugging task. Without `JTAG` support in the default USB-JTAG module, the main method of debugging can only be done by having `RS232` test print messages. There is no full code visibility on the target, as compared to the case for a `LINUX` host environment.

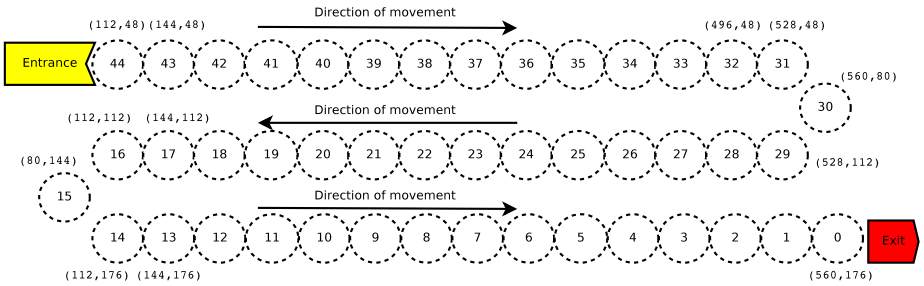


Figure 4.1: Pixel locations and index of moving balls on a VGA screen

4.2 Game Algorithm

Ang Lay Hong, Chaiwat Sittisombut

The following section describes briefly about the Zuma game algorithm implemented.

4.2.1 Placement of Moving Sequence Balls on VGA Screen

Ang Lay Hong

Information of sequence balls (objects) are stored in array of data structure, and indexed in order with the ball nearest to the exit as 0 and last ball being nearest to the entrance. Each data structure contains information about the pixel location of the ball on screen, given by the variables `xy_index`, `x` and `y`. The on-screen size of each ball is 32 by 32 pixels.

The variable `xy_index` gives a label to the location of a moving ball on screen and `x` and `y` stores the pixel location. There are 45 index locations (0 to 44), with pre-allocated (x, y) pixel coordinates, representing the center of the ball located at the index position. Index 0 (with coordinates $(560, 176)$) represents the ball center location nearest to the exit and 44 (with coordinates $(112, 48)$) being the index next to the entrance, as shown in Figure 4.1.

The movement of sequence balls are simulated by moving 2 pixels once every few ms. When that happens, the centre (x, y) is updated, depending on direction of movement of the ball. When the balls are moved 32 pixels away from its original center (x, y), it corresponds to the next ball locaiton. Its associated `xy_index` is then updated to the next value.

It is best to illustrate the above concept with an example. Say, given 4 generated sequence balls with locations as shown in Figure 4.2a. The data structure for these 4 objects are

```
ball[0] {xy_index = 14; x = 112; y = 176;}
ball[1] {xy_index = 15; x =  80; y = 144;}
ball[2] {xy_index = 16; x = 112; y = 112;}
ball[3] {xy_index = 17; x = 144; y = 112;}
```

If the balls are moved by 20 pixels, the new locations of the balls, shown in Figure 4.2b, become

```
ball[0] {xy_index = 14; x = 132; y = 176;}
ball[1] {xy_index = 15; x = 100; y = 164;}
ball[2] {xy_index = 16; x =  92; y = 132;}
ball[3] {xy_index = 17; x = 124; y = 112;}
```

When the balls are moved further by 12 pixels, the moving balls are updated as shown in Figure 4.2c. Note that the value `xy_index` has changed.

```
ball[0] {xy_index = 13; x = 144; y = 176;}
ball[1] {xy_index = 14; x = 112; y = 176;}
ball[2] {xy_index = 15; x =  80; y = 144;}
ball[3] {xy_index = 16; x = 112; y = 112;}
```

4.2.2 Keyboard Control

Ang Lay Hong

The player of the Zuma game controls the shooter ball via arrow keys on a PS/2 keyboard. The `LEFT` and `RIGHT` keys control the horizonal movement of the shooter ball/stand across the screen. Even when the shooter ball is launched and still in trajectory towards the sequence of colored balls, the player is still able to control movement of the shooter stand via the `LEFT` and `RIGHT` keys. When the `SPACEBAR` key is hit, the shooter ball is fired towards the sequence of colored balls. Hitting the `ENTER` key restarts the game when game over, or starts the game at next level of difficulty if the player has successfully destroyed all the balls on the screen.

4.2.3 Collision Detection Conditions

Chaiwat Sittisombut

After a shooter ball is launched, the software system constantly checks for any collision between the shooter ball with any of the sequence balls.

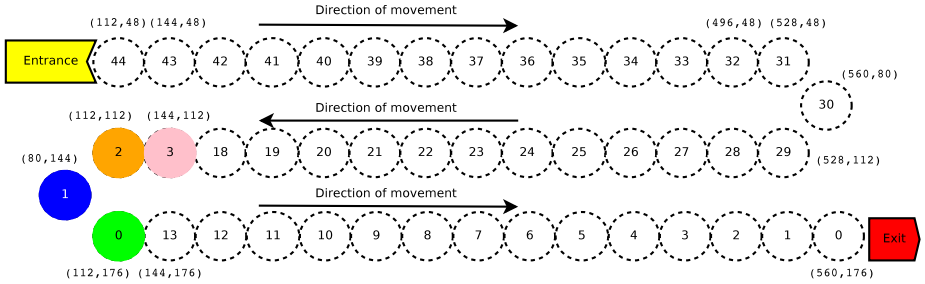
To determine a collision, given n moving sequence balls on screen and the coordinates of shooter ball being (x_{shoot}, y_{shoot}), a collision is detected when the following conditions are fulfilled.

$$\begin{aligned} (x_n - 32) < x_{shoot} < (x_n + 32), \\ \text{and} \qquad y_{shoot} \leq (y_n + 32) \end{aligned}$$

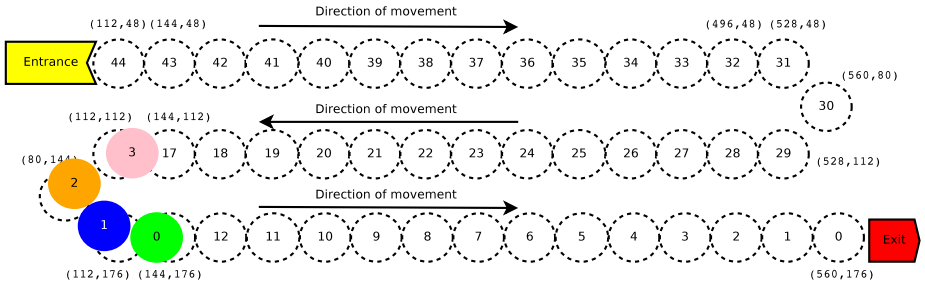
When there are more than one sequence balls detected to fulfill the above condition, the sequence ball nearer to the shooter ball is considered as the collided ball. If the shooter ball collides right in the middle of two sequence balls, the collision is set to the sequence ball nearer to the exit. Figure 4.3 illustrates graphically the collision area check.

4.2.4 Collision Handling Conditions

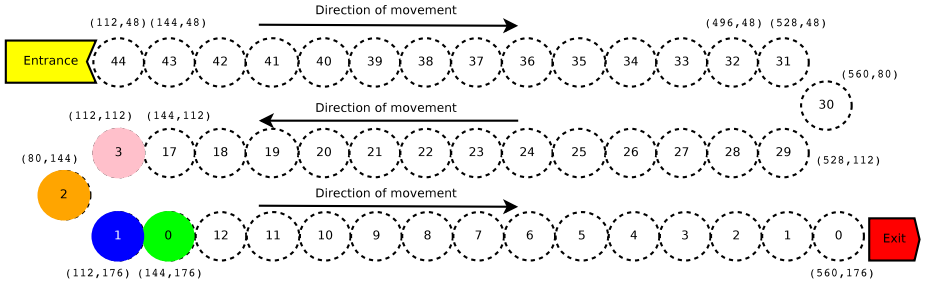
Ang Lay Hong



(a)



(b)



(c)

Figure 4.2: (a)Example of 4 moving balls on a VGA screen, (b)New pixel locations after moving 20 pixels, (c)New pixel locations after moving another 12 pixels.

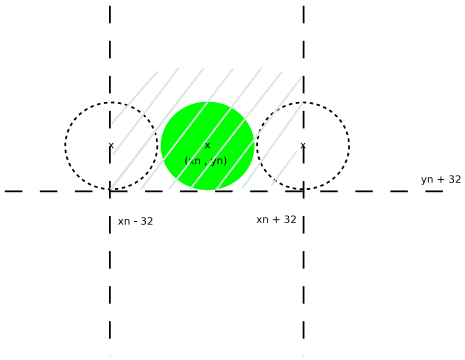


Figure 4.3: Collision check boundary condition

When collision is detected, various condition checks are made to determine if an explosion has occurred.

Let index of the sequence ball that collided with shooter ball be n. One of the following cases may happen.

Case 1 *The shooter ball has the same color as ball n. In this case, check for consecutive colored balls from ball n downto 0 and n to $(N-1)$, where N is the number of balls in the sequence. See Figure 4.4a.*

Case 2 *The shooter ball has a different color from ball n. The shooter ball is nearer to ball $(n-1)$ than ball $(n+1)$. In this case, if the shooter ball has the same color as ball $(n-1)$, check for consecutive colored balls from ball $(n-1)$ downto 0. See Figure 4.4b.*

Case 3 *The shooter ball has a different color from ball n. The shooter ball is nearer to ball $(n+1)$ than ball $(n-1)$. In this case, if the shooter ball has the same color as ball $(n+1)$, check for consecutive colored balls from ball $(n+1)$ to $(N-1)$. See Figure 4.4c.*

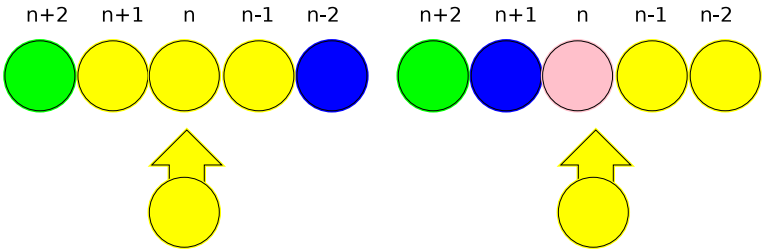
For each of the cases above, the chain length of identical colored balls, including the shooter ball, is determined. If the chain length is at least 3, these colored balls are destroyed. The remaining sequence balls are then collapsed and checked for further collisions based on the following condition.

Let $(n_{head}+1)$ and $(n_{tail}-1)$ be ball index of first and last exploded balls in the sequence as shown in Figure 4.5a. If balls n_{head} and n_{tail} has the same color, then check for balls from n_{head} downto 0 and n_{tail} to $(N-1)$, where N is the length of sequence balls.

If the above case form chain length of 3 or more colored balls, the second explosion and further collapsing occurs, as shown in Figure 4.5b. The above check continues until there are no further explosions or all balls in the sequence are destroyed.

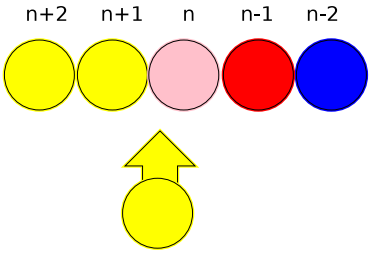
4.2.5 Shooter Ball Insertion

Ang Lay Hong



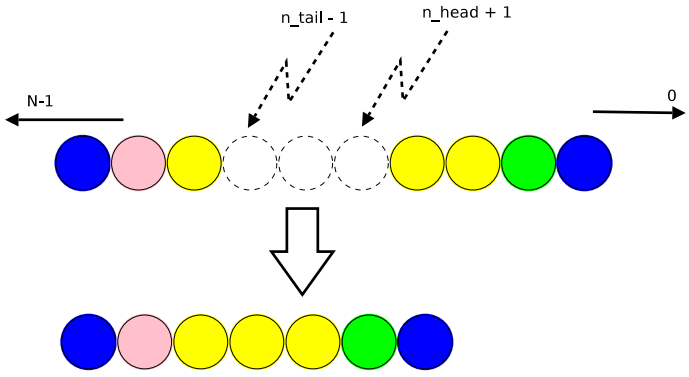
(a) Case 1: shooter ball hits the same colored ball n

(b) Case 2: shooter ball hits the same colored ball $n-1$

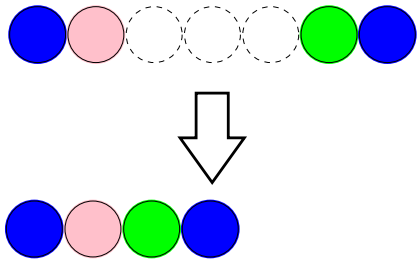


(c) Case 3: shooter ball hits the same colored ball $n+1$

Figure 4.4: The 3 possible cases for the shooter ball to destroy the sequence balls



(a) Sequence of balls after first collision



(b) Sequence of balls after second collision

Figure 4.5: An illustration of the second collision

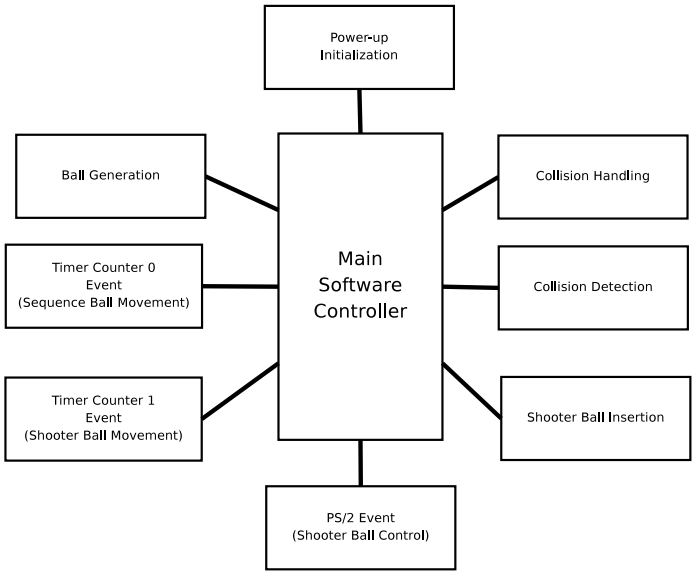


Figure 4.6: Software Modules in Microblaze

When no 2 or more sequence balls carry the same color as the shooter ball, the shooter ball is inserted into the sequence of moving balls. The shooter ball is inserted based on the following conditions.

Case 1 *If shooter ball hits directly on sequence ball n_i, move sequence balls* 0 *to n_i one index position closer to the exit. Insert shooter ball in pixel location occupied by n_i.*

Case 2 *If shooter ball hits 2 sequence balls n_i and n_{i+1}, move sequence balls* 0 *to n_i one index position closer to the exit. Insert shooter ball in pixel location occupied by n_i.*

Case 3 *If shooter ball hits the last ball in the sequence, n_{L-1}, move sequence balls* 0 *to n_{L-1} one index position closer to the exit. Insert shooter ball in pixel location occupied by n_{L-1}.*

Case 4 *If shooter ball hits the first ball in the sequence, insert shooter ball in one index position before sequence ball* 0.

4.3 Software Modules and Features

Ang Lay Hong, Chaiwat Sittisombut

The various software modules implemented in the Microblaze are shown in Figure 4.6.

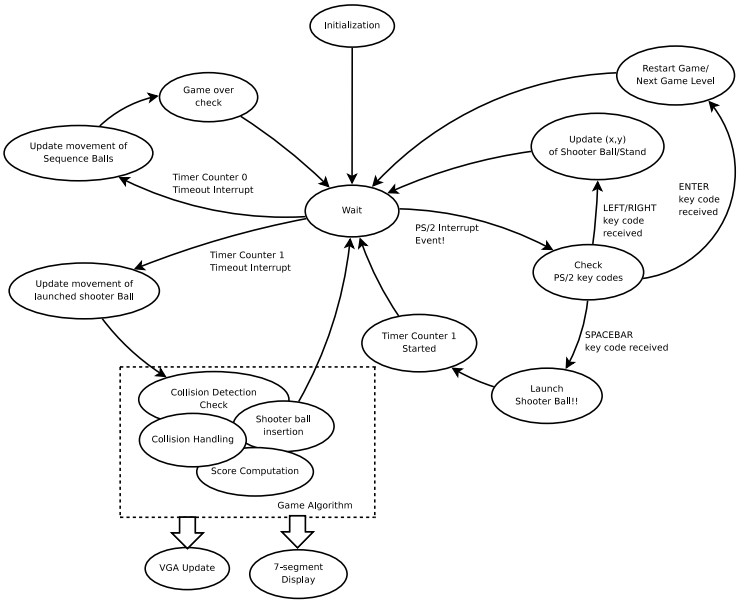


Figure 4.7: State Flow Diagram for the Software Main Controller

4.3.1 Main Controller

Ang Lay Hong

The main controller is responsible for handling of all tasks based on events of occurrance. A state flow diagram in Figure 4.7 illustrates issuance of tasks based on various conditions.

4.3.2 Generation of Sequence Colored Balls

Ang Lay Hong

The sequence of colored balls is generated once every few milliseconds, using a random number generating function as described in section 4.3.4. The generation is performed until the defined maximum number of colored balls for a level is reached. The number of different colors is determined by the level of difficulty. For game level 1, the generated balls carry 3 different colors. The number of different colors then increments with level of difficulty, up to 8 different colors.

The sequence of balls is configured to move 2 pixels every $100ms$. When the last ball in the sequence (i.e. the one closest to the entrance) has (x, y) coordinates of of $(110, 48)$, the next ball is generated but "hidden" behind the entrance box.

4.3.3 Generation of Shooter Ball

Ang Lay Hong

A new colored shooter ball is generated after the shooter ball collides with the sequence of balls and all chain reactions are completed. The color generated for the shooter ball has to be one of the colors in the existing sequence of moving balls.

4.3.4 Pseudorandom Number Generator

Ang Lay Hong

A random number generator (RNG) is required to randomize the colors of the sequence of balls as well as the shooter ball. To do so, the `libc` random number function `rand()` is considered. It is however observed that the `rand()` function halts the interrupt controller. In addition, the `libc` function takes up a ridiculously huge code size of 8 kB. Due to the above motivation, a separate Pseudorandom number generating (PRNG) function is required. A simple alternative is the `Rnd()` function implemented in Visual Basic, which is a linear congruential generator. However, due to its lack of randomness, it is not recommended to use the PRNG for other purposes, except for trival games.

The formula for `Rnd()` function is given as

$$x_1 = (a \cdot x_0 + c) \bmod 2^{24} \quad \text{for} \quad \begin{array}{l} a = 1140671485, \\ c = 12820163 \end{array}$$

The term x_1 denotes the random number generated based on the previous generated random number x_0. The default seeding value for x_0 is 327680

The color generated is represented by a number from 0 to `NUM_OF_COLORS`-1. This is given as `NEW_COLOR` $= x_1 \bmod$ `NUM_OF_COLORS`

4.3.5 Motion of Moving Balls on VGA screen

Ang Lay Hong

Game algorithm requires constant movement of the sequence of colored balls at a specific rate. This is handled by the timer counter 0 which is interrupted once every defined period of time and the (x, y) coordinates of the balls are then updated. The timer reset value is defined as $12,500,000$, which corresponds to 250 ms. The sequence balls are configured to move 2 pixels every 250 ms. Figure 4.8 demonstrates the effect of sequence ball movement being handled by timer counter 0.

When the shooter ball is fire, it moves vertically towards the sequence of colored balls at a specific rate. In order to simulate the movement, timer counter 1 is used. The timer counter 1 is interrupted once every 10 ms to update the location of the shooter ball by 16 pixels vertically. Figure 4.9 illustrates the motion of the shooter ball with respect to the period of timer counter 1 when launched.

4.3.6 Keyboard Control of Shooter Ball

Ang Lay Hong

The player may move the shooter ball horizonatally across screen to aim the shooter ball at the sequence of balls. This is controlled via the `LEFT` and `RIGHT`

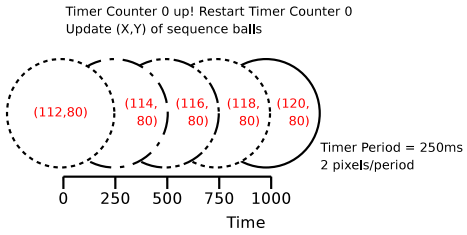


Figure 4.8: Motion of sequence of balls with respect to Timer Counter 0

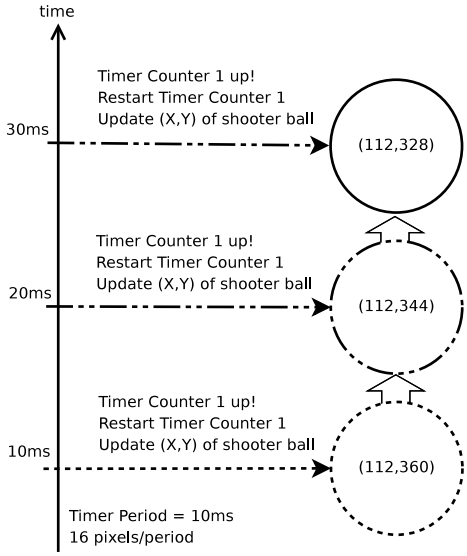


Figure 4.9: Shooter ball motion with respect to Timer Counter 1

arrow keys on the keyboard. The first scan key is received as a PS/2 interrupt signal when the `LEFT` or `RIGHT` arrow key is pressed. The interrupt event will trigger the software module to update the x coordinate of

- the shooter ball and shooting stand if ball is unlaunched
- the shooter stand if ball has been fired

As the player may hold down the arrow key for consecutive left or right movement. The scan code sequence for such event will be

```
LEFT:  E0 6B E0 6B ... E0 6B E0 F0 6B
RIGHT: E0 74 E0 74 ... E0 74 E0 F0 74
```

The key scan sequence is handled by receiving only the first 2 scan codes via interrupt. The (x, y) of the shooter ball/stand is updated and the main controller goes into polling mode for the next PS/2 byte. When the next 2 bytes of `E0 6B` or `E0 74` are received during the wait loop, the update is performed again. Receiving the key codes in such single byte polling mode frees up the system for more important events such as updating of movement of sequence balls and shooter ball. In addition, it is expected that the PS/2 polling function will not hog the system as key codes are expected to come in intervals of 100 ms before the echo code `F0` is received.

The shooter ball is launched from the shooting stand via the `SPACEBAR` key. When the PS/2 interrupt or polling module detects the press of `SPACEBAR` key, timer counter 1 is started to update the vertical movement of the shooter ball as described in previous section. If the shooter ball is already launched and in trajectory towards the sequence balls when the `SPACEBAR` key is pressed, the system does not respond to the key pressed.

During game play, if the `ENTER` key is hit, the system ignores the scan code. Upon completion of a game level or when the gameplay is over (first colored ball hitting the exit), the `ENTER` key is enabled. Hitting the `ENTER` key brings the game to the next level of difficulty or re-start the game by generating a new sequence of colored balls.

4.3.7 Collision Detection Module

Chaiwat Sittisombut

The collision detection module is executed in a timely manner after the shooter ball has been fired from the shooter stand. The module is called periodically, with its period controlled by Timer Counter 1. At every timeout, the module performs a collision check. If collision occurs, the index n of the ball at which the collision occurs is determined.

Given `y_pos` as the y-coordinate of the shooter ball. The condition `y_pos`< 224 is first checked. The y-coordinate range of 224 to 479 is the region below the path of the moving balls, of which no collision is likely to occur.

If the condition `y_pos`< 224 is fulfilled, boundary conditions as described in section 4.2.3 are checked.

To perform the boundary check, the horizontal and vertical distances between the shooter ball and each of the sequence balls starting from the first

ball are computed and checked. When the boundary conditions are met for a sequence ball index, a collision is detected and Timer Counter 1 is stopped. The check stops and the ball index is considered as the collided ball index n. The collided ball index however may not be the ball closest to the shooter ball, as the next sequence ball is unchecked yet. To do so, the distances between the shooter ball and the next sequence ball are also computed and compared.

Let the xy-coordinates of shooter ball be (x_s, y_s) and the 2 checked sequence balls be (x_i, y_i) and (x_{i+1}, y_{i+1}) respectively.

If $|x_s - x_{i+1}| \geq |x_s - x_i|$, the shooter ball is closer to the currently checked sequence ball index i. If the reverse is true, the collided ball index is set as the next ball index, i.e $(i + 1)$.

4.3.8 Collision Handling Module

Chaiwat Sittisombut

The collision handling module performs the check on color chain collision, given the collided ball index n determined in section 4.3.7. When 3 or more balls with the same color collide (including the shooter ball), ball destruction is handled. Further chain reaction checks are made to detect multiple chain explosions.

The module may be separated into 3 cases as described in section 4.2.4. The cases are determined based on the color check between the shooter ball and the collided ball.

If the shooter ball and the collided ball have the same color, case 1 is considered. The length of chain color is determined. A subroutine is implemented to compute the length of chain with the collided index n as the "seed" ball index. The subroutine returns 3 variables `chain_length`, `index_start` and `index_end`. These are the chain length and the starting and ending index of the color chain, respectively.

If `chain_length`≥ 3, an explosion has occurred and the sequence of balls are collapsed and updated with the collided balls removed. However, if the `chain_length`< 3, there is no explosion and the shooter ball is inserted into the sequence of moving balls.

After first collision has occurred, the next collision in a new ball sequence must be detected with the same subroutine, using the information in `index_start` and `index_end`. Nevertheless, if there is any collision that occurs in the beginning or in the end of the ball sequence, there is no need for further collision checks. If there are more than 2 multiple collisions, a bonus score is happily given. This repeats until all subsequent chain reactions are completed or all sequence balls are destroyed.

The multiple collision detection and handling have the same principle as the single collision detection and handling. However, a new variable n_chains is introduced to count the number of collisions which has just occurred. Moreover, the collided balls are removed and the closest ball to the shooter ball or a previous ball index n, is set to be a new ball index m. The new ball index is used for comparing color with the adjancent ball.

If the shooter ball and the collided ball have a different color, cases 2 and 3 are checked, where the shooter ball will be checked for color chain collision against one of the 2 neighboring sequence balls.

To determine the closer of the two sequence balls to the shooter ball, the distances are computed as follows.

Let the x-coordinate of shooter ball be x_s and that of the 2 neighboring sequence balls be x_{n-1} and x_{n+1} respectively.

If $|x_s - x_{n+1}| \geq |x_s - x_{n-1}|$, the shooter ball is closer to the sequence ball $(n-1)$. Chain collision is thus checked from ball index $(n-1)$ onwards downto first ball in the sequence. If the condition fails, the shooter ball is found to be closer to the sequence ball $(n+1)$. Instead, chain collision is checked from ball index $(n+1)$ until end of ball sequence. Just as in case 1, further chain collision are checked if a chain collision is detected.

If all the above 3 cases are invalid, i.e. the shooter ball has a different color from ball n, $n-1$ or $n+1$, the shooter ball is inserted into the ball sequence.

To simulate the effect of collision on screen, the VGA display screen will be updated immediately when an explosion occurs. Timer Counter 1 is then used to wait for approximately 0.5 sec before the next check is performed. This pause is to simulate effect of ball destruction on the VGA screen.

4.3.9 Ball Insertion Module

Ang Lay Hong

The ball insertion module is called when there is no destruction of colored balls and the shooter ball is to be inserted into the existing sequence of moving balls, taking over the pixel location of ball n.

Let N be the number of current moving balls on screen. The balls from $(n+1)$ to the last ball $(N-1)$, i.e. nearest to the entrance, will retain their current local pixel location. Their ball indexes will however be increment since the number of balls will be incremented to $(N+1)$ after the shooter is inserted. Thus, these balls will be represented with ball indexes $(n+2)$ to N.

The next step is to move balls n to 1 one position closer to the exit. This is equivalent to taking over the information of `xy_index`, `x` and `y` from the ball in front. For instance, ball $(n-1)$ will be updated with the pixel location of ball $(n-2)$, $(n-2)$ with $(n-3)$ and so on, until ball 1 being updated with pixel location of ball 0. As for ball 0, extra care is taken to determine its position based on its current location. Its new pixel location is computed based on the current offset of (x,y) from the pre-allocated `(x,y)` for `xy_index`.

After balls 0 to n are moved ahead, the shooter ball is inserted as ball $(n+1)$ by taking over the `xy_index`, `x` and `y` of ball n.

4.3.10 VGA Controller Update

Chaiwat Sittisombut

The VGA controller core described in section 3.2 paints the screen based on the center pixel locations and color information received from Microblaze. To facilitate transfer of such data, fifty 32-bit registers are used. Microblaze writes to these slave registers via memory-mapped addressing method.

The format of each of the 32-bit register is given as follows.

Bits	31	30:27	26:19	18:10	9:0
Length	1 bit	4 bits	8 bits	9 bits	10 bits
Data	Valid bit	Reserved	RBG color	center Y pixel	center X pixel

Upon any changes to the locations of the moving and shooter balls on screen, as well as the colors, these registers will be updated with the latest data.

4.3.11 7-Segment Display Score Update

Chaiwat Sittisombut

The 7-segment display core described in section 3.1 displays the current score for the game. The score is displayed in four display segments, D_0 through D_3, for the $1,000th$, $100th$, $10th$ and ones places respectively.

Scores are written into one 32-bit register in the 7-segment display, via memory-mapped address accessing mode.

The format of the 32-bit register is

Segment	D_3	D_2	D_1	D_0
Bits	31:24	23:16	15:8	7:0
Place	$1000th$	$100th$	$10th$	ones

The 7-segment display core is only capable of displaying digits 0 to 9. Therefore, the score is converted from hexadecimal to decimal numbers for each digit, before writing to the 32-bit register. The conversion can be illustrated by the following example.

Given `score`= 1826, then

$$\begin{aligned} D_3 &= \lfloor \tfrac{1826}{1000} \rfloor \bmod 10 &= 1 \\ D_2 &= \lfloor \tfrac{1826}{100} \rfloor \bmod 10 &= 8 \\ D_1 &= \lfloor \tfrac{1826}{10} \rfloor \bmod 10 &= 2 \\ D_0 &= 1826 \bmod 10 &= 6 \end{aligned}$$

4.3.12 Power-Up Initialization

Ang Lay Hong

Upon starting of the game, the software performs a series of events at initialization, as illustrated in Figure 4.10.

The pseudorandom number generator (PRNG) described in section 4.3.4 is initialized with a default seed value of 327680.

A new sequence of colors are generated using the PRNG initialized. The starting (x, y) coordinates of the colored balls are assigned based on pre-adefined locations specified in a constant array.

A color is generated for the new shooter ball and the initial locaiton for the shooter ball is set to be the lower centre of the scren, i.e. $(x, y) = (320, 430)$.

The timer counter 0 device, which handle the movement of the sequence ball, is configured with a pre-defined period. The defined period is 250 ms, based on

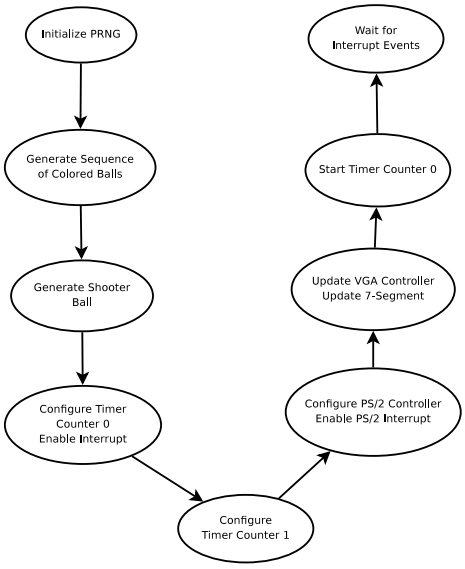


Figure 4.10: Sequence of events during power-up initialization

a 50 MHz clock. This gives a counter reset value of 12, 500, 000. The timer counter 0 is also configured to be interrupt driven, via the interrupt controller.

The timer counter 1 device, which will be used when the shooter ball is fired during the game, is configured with a reset value of 500, 000, signifying a period of 10 ms.

The PS/2 controller that tracks the player control of the shooter ball is configured to be interrupt driven, via the interrupt controller. The interrupt is enabled to start detecting any key press interrupt activity from the PS/2 keyboard.

With the new sequence of colored balls and shooter ball ready, the VGA controller registers are updated with the details of the colored balls. The scoreboard is reset to zero and the 7-segment display is updated with the reset score.

Finally, timer counter 0 is started to start handling the movement of sequence colored balls and the software goes into a wait state, as described in section 4.3.1.

4.4 Memory Utilization

Ang Lay Hong

The memory utilization for code and data in the Microblaze is 22kB of BRAM.

4.5 Problems and Issues

Ang Lay Hong, Chaiwat Sittisombut

The following sections describes the various problems and issues encountered during the design and implementation of the software architecture in Microblaze. Workarounds and solutions to these problems and issues are also presented.

4.5.1 Code and Data Memory

The SPARTAN3E-1200 FPGA chip on the Nexsys2 kit comes with an internal block RAM (BRAM) memory of 507 kb (i.e. 63 kB). However, only up to 32 kB is usable for code and data in Microblaze. In order to eliminate the issue of limited code size, there were plans to reside the code and data in the external SRAM memory instead. However, it is observed that if SDRAM is used for data and code memory, Xilinx debugging tools are required to load the data and code into SDRAM, which is not supported by the on-board USB-JTAG connection. Thus, code and data are instead residing on BRAM.

4.5.2 Avoiding `libc` Library Function Calls

It is crucial to avoid using `libc` library function calls in the software module. such as `printf` and `rand`. The code size increases dramatically due to the need to include the `libc` library.

As described in section 4.3.4, using `rand` as a PRNG affects other features such as the interrupt controller.

4.5.3 Software Debugging

Ang Lay Hong, Chaiwat Sittisombut

Software debugging on a Microblaze environment without the help of a Xilinx JTAG tool is a challenging task, as there is no code visibility for tasks running on the Microblaze. One solution to this problem is to first perform all possible testing, debugging and simulations of the game algorithm in the LINUX host environment. The software is then ported to Microblaze platform for further verification on the Nexys2 target board. Bugs discovered on the target platform can be easily resolved by simulating them on the LINUX platform. Extra care has to be taken during the porting as several function calls such as `printf()` and `rand()` are not usable, as described in sections above.

Chapter 5

Integration & Testing

Ang Lay Hong, Chaiwat Sittisombut, Lim Wee Guan

This chapter describes the porting of the host version of the software, and the integration of the hardware components to the board.

5.1 Software Porting

Ang Lay Hong

For a more efficient software development process, the game algorithm is first implemented and tested on the LINUX host platform. With the `gdb` debugging environment, software issues are more easily resolved.

The target software framework, consisting of necessary target-dependent modules such as the 2 interrupt-driven Timer Counters and PS/2 controller, are developed on Microblaze target.

Having done all necessary simulation and testing on the `LINUX` platform, the game module is then ported to the Microblaze target platform and incorporated into the target software framework. Before porting, the target software framework is first tested thoroughly. This is to ensure that issues can be more easily resolved during the integration. It eliminates unnecessary guesswork required in determining if the game algorithm module or the framework creates the issues.

Further testing are then performed with the target software, while the hardware is in development. In the absence of a `JTAG` cable, `RS232` test print messages are used to facilitate the debugging process.

5.2 Hardware Interfacing

Lim Wee Guan

This section describes the tasks that needed to be performed to interface the custom VHDL blocks to the Microblaze. The initial steps are done in the XPS Wizard that creates a `user_logic.vhd` and a `custom_core.vhd` in the `pcores\custom_core\hdl\vhdl\` directory.

The next step is to customize the two VHDL files, essentially to update the `entity` to incorporate the additional user files and instantiating any of the custom VHDL blocks. It may also be necessary to modify the Xilinx code in the `user_logic.vhd` file that allows the user registers to be read by the custom logic. In the case of the VGA Controller, the large number of user registers (50) meant that the Xilinx logic in `user_logic.vhd` takes up a large amount of resources. By removing the registers, it was possible to free up FPGA resources, but the draw-back is that the Microblaze is unable to read back data that has been written to the user registers. This is not a big problem, as all data in this application is highly dynamic and changes are written into the registers regularly without the need to read back any values.

The last step in the interfacing is to modify the `.mpd` and `.pao` files in the `pcores\custom_core\data\` directory to inform XPS about the user ports in the custom logic and the file synthesis order respectively. After all this, XPS needs to be told to read the changed `.mpd` and `.pao` files by re-importing the peripheral through the XPS Peripheral Wizard.

As described in Section 3.2.1, it is important for the custom logic to be tested in a standalone fashion prior to attempting to interface it with the Microblaze. This will ensure that one variable (the custom VHDL code) is fixed before changing the other variable (Microblaze interface). Another limitation of the XPS is that it has very limited support for simulation e.g. ModelSim and thus troubleshooting the interfacing with Microblaze is tricky enough without having non-deterministic VHDL blocks being added into the equation.

The last stumbling block faced in the project is that the XPS environment is rather software development oriented in contrast with the hardware oriented ISE. Thus, it was found that ModelSim/ISE is much more suitable for hardware development than working within XPS. This however, is a shame, as two independent workflows, one for hardware and one for software does not make the environment optimal for integrated development of hardware and software.

5.3 Testing

Chaiwat Sittisombut

All three user interfaces (Keyboard, Monitor and 7-segment) and the game algorithm were integrated and tested. It can be concluded as following:

A *104-key IBM compatible PS/2 keyboard* was used during the test. The `LEFT` and `RIGHT` keys in the cursor control are functional. However, controlling the shooter ball via the `LEFT` and `RIGHT` keys on the numeric keypad causes the shooter ball move to the far left or right.

A 24" *Sun Microsystems* VGA monitor and a 19" *HP* TFT monitor were tested to display the game. The game can be displayed on both monitors without any problems.

The four 7-Segment displays on Nexys2 board displayed the 4 digits properly and gave results similar to that in the game algorithm (viewed from UART). The game could be played for more than an hour without any unpredictable bugs.

Chapter 6

Conclusion

Ang Lay Hong, Chaiwat Sittisombut, Lim Wee Guan

This project was done in fulfillment of the course requirements for EDA385, Embedded Systems Advanced Course. To quote from the Course website, this course aims to allow students to "complete their knowledge of embedded systems acquired during the basic Embedded Systems Design course (EDA380) with practical experience. More often than not, going from theory to practice is so demanding that students without any practical experience feel lost even having a good theoretical background. This course will give you the essential experience required to fill in this gap."

In addition to the above stated objective of gaining practical development experience, this course also gave the opportunity to obtain real-world insights in working on embedded systems projects, complete with tight deadlines and having to work as a team with multiple members, some of whom have agendas that differ from the rest of the group's objective and goals. That said, it was fulfilling to be able to define, prototype and build a system from scratch, all in six weeks.

6.1 Lessons Learnt

The key lessons learnt in the project is that the traditional development methodology of building the hardware and then moving on to building the software once the hardware platform is ready/stable does not work for projects with tight deadlines such as this. Some form of concurrent hardware and software development is necessary else either party will be waiting for the other, and the project deadline will not be met.

It was also realized that integration and testing takes up a sizeable amount of time, and is often the key determining factor for success/failure in the development of the system. This is especially true in the case of embedded system integration where both the hardware and software are being integrated are unstable.

Bibliography

[1] Zuma (video game). http://en.wikipedia.org/wiki/Zuma_(video_game), Nov 2008.

[2] Digilent Nexys2 board reference manual. http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf, June 2008.

[3] Nexys base system builder guide for EDK. http://www.digilentinc.com/Data/Products/NEXYS2/Digilent_Nexys_Board_Support_Package_V_1_20.zip, Mar 2008.

[4] Adam Chapweske. The PS/2 mouse/keyboard protocol. http://www.computer-engineering.org/ps2protocol/, May 2003.

[5] IBM Corp. *128-Bit Processor Local Bus Architecture Specifications*, May 2007. Version 4.7.

[6] Xilinx Inc. *Embedded System Tools, Reference Manual*. EDK 10.1, Service Pack 2.