

ECE 532 “PONG” Group Report

Chirag Ravishankar (995399108)

Durwyn D'Silva (994761496)

Jeffrey Goeders (993367566)

April 5, 2010

Contents

- 1 Overview 3
 - 1.1 Goals 3
 - 1.2 Background 3
 - 1.3 System Overview..... 3
 - 1.4 IP and Hardware Descriptions..... 5
- 2 Project Outcome 7
 - 2.1 Our Final Result and Key Successes 7
 - 2.2 Shortcomings 7
 - 2.3 Future Improvements 7
- 3 Modular Descriptions..... 9
 - 3.1 Video to Memory Module..... 9
 - 3.2 Paddle Detector Module..... 13
 - 3.3 Paddle and Ball Drawer Program 17
 - 3.4 Sound Module..... 19
 - 3.5 Ball and Score Control Module 22
- 4 References 30
- 5 Appendix 31
 - 5.1 PLB Master, Burst Write Operation 31
 - 5.2 PLB Master, Burst Read Operation 32

1 Overview

1.1 Goals

The goal of our project was to recreate the classic 2-player game, Pong, with added special features. The vision was to have the players hold and move physical paddles hit to a virtual ball being projected onto the screen. Image detection would be performed to detect the paddles, and calculations in hardware would detect collisions, as well as the velocities and angles of the paddles. The game would be played on a projected screen.

1.2 Background

Pong is one of the oldest and most popular arcade games. It is a digital version of tennis where two players have a racquet/paddle on either side of the screen and the object of the game is to hit the ball and score on the opponent. We wanted make this game more interactive such that the players have physical paddles in their hands and using image detection, we would detect and draw them on a screen. A ball will be drawn on the screen and the players have to move their paddles to interact with the ball. Contrary to the classic game where the paddles are constrained to move only up and down, our game would allow the players to move it in any direction. Calculations based on the angle and velocities would be done to correspondingly interact with the ball



Figure 1: Classic Pong Game (left) and our version with the XUP-V2P Board(right)

1.3 System Overview

Our system diagram is similar to that of our proposal. However, the paddle and ball drawer was implemented in software instead of on a separate hardware core. We also added an audio module to play distinct sounds corresponding to the gameplay.

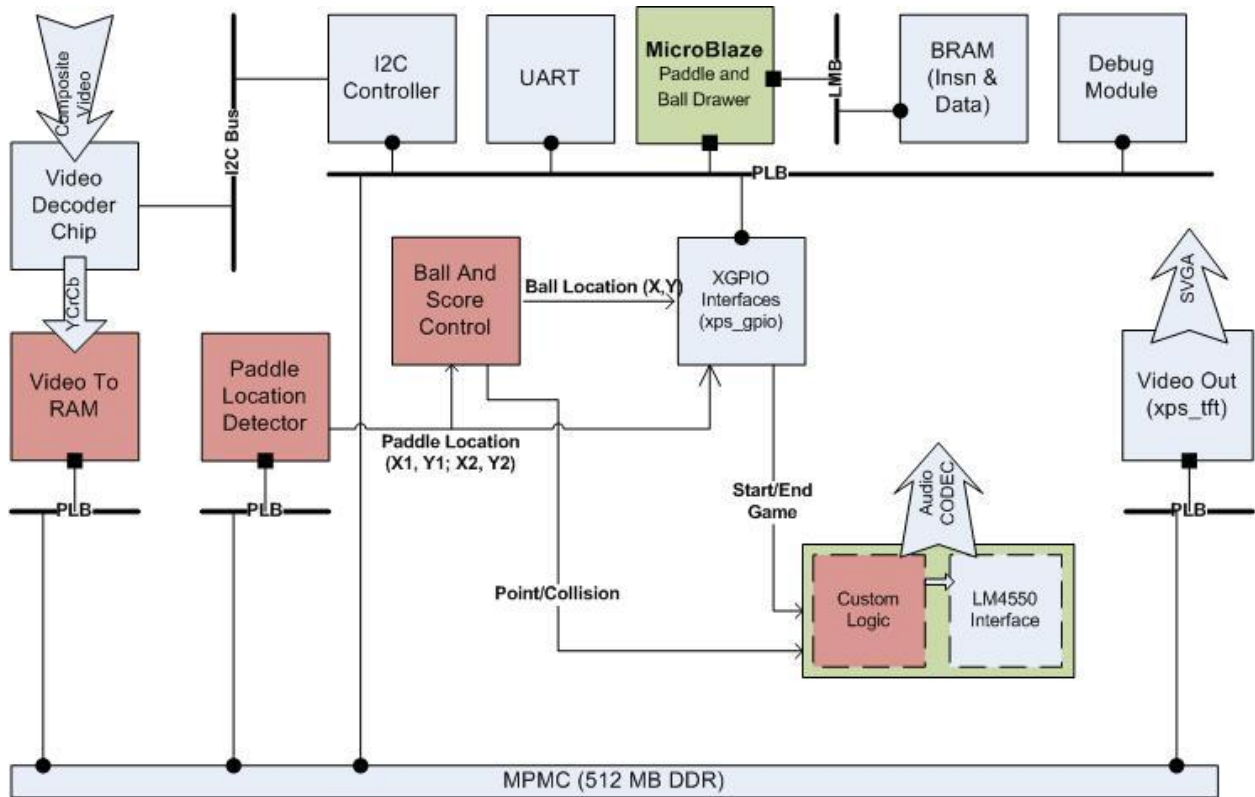


Figure 2: System Diagram

1.4 IP and Hardware Descriptions

IP	Function	Author
IP inside the FPGA		
video_to_ram	The core captures video from the camera and inputs the information in the correct format to the DDR-SDRAM through the PLB	Group
paddle_detector	Analyzes the video data to find points of a specific colour as specified by the MicroBlaze program	Group
ball_control	Using the paddle locations, creates a ball location and identifies collisions between the paddles and the ball. Also does the physics calculations of the angles and velocities. This module also keeps track of the player scores	Group
audio_core	Interface with the LM4550 chip to play sounds	Embedded Computing
Audio_core custom logic	Play 4 distinct sounds based on the status of the game	Group
Microblaze	Processor core for setting up video, drawing the paddles and ball, interface with the UART	Xilinx; program implemented by the group
dlmb/dlmb_ctrl	Data memory controller interfaced through Local Memory Bus	Xilinx
ilmb/ilmb_ctrl	Instruction memory controller interfaced through LMB	Xilinx
PLB (plb_v46)	Processor Local Bus used to interface to various IP cores including the microblaze, video_out, paddle_detector and video_to_ram	Xilinx
XGpio (xps_gpio)	Used to acquire and write signals from/to	Xilinx

	microblaze and other IP cores	
IIC (xps_iic)		Xilinx
Debug Module (mdm)	Debug Module to enable XMD	Xilinx
IP outside the FPGA		
DDR_SDRAM (mpmc)	Memory to hold the video data, Multi-ported memory controller is used to interface to several cores	Xilinx
LM4550	Audio Codec chip	National Semiconductor
Video Daughter Card	Interface with the video camera	Analog Devices
Video Camera	Capture the video of the players with paddles	
Speakers	Output audio data corresponding to the game	
UART	Debug information; report player scores and display status of the game	Xilinx
VGA monitor/Projector	Display the paddle and the ball	

Table 1: Description and Source of all components of the design.

2 Project Outcome

2.1 Our Final Result and Key Successes

We have successfully implemented our project. We have implemented successful video capture, storage of the frames to RAM, ability to decode the frame and detect the paddles, perform physics calculations to determine the interaction between the paddles and a virtual ball, and draw the paddles and ball on the screen. The system implemented is the same as the diagram shown above.

2.2 Shortcomings

A shortcoming of this project is that our decision to implement the Physics in hardware rather than software caused us difficulties and delayed the project. There are several advantages to software we realized in the later portion of our project:

1. **Turn Around time** – Compilation time for the hardware implementation (synthesis + place and route + Bit stream) were considerably longer than software compilation times (Compiler + Bit stream).
2. **Availability of IPs to calculate Math** – The major problem with the hardware implementation was the availability of IPs such as the “Cordic” Math IP and Divider IPs. In order to implement the physics, we required trigonometric functions (Sin/Cos) as well as simple math functions like Divide. We spent a large amount of our time getting the IPs. In the end, we abandoned using these IPs.

In addition to the physics, the ball detector core does not automatically calibrate to lighting conditions and the paddle/ball drawer software currently causes the paddle and ball to “flicker” by printing alternating white and black pixels frequently.

Another shortcoming is the proper detection of the location of the paddles. Sometimes, other items seen by the camera are interpreted to be the paddle ends and the coordinates of the paddle are updated to be incorrect. This causes the paddle to jump to unexpected places on the screen.

2.3 Future Improvements

We could improve the physics by implementing a software based angle detection scheme. For example, we could implement a second Micro Blaze system which has the sole purpose of calculating the angle of

intersect between a paddle and the ball. This would allow us to achieve more realistic results when collisions occur.

In addition, the paddle/ball drawer software on the existing micro blaze could be tweaked so that no flicker is present and we could implement calibration routines in the ball detector core.

In terms of better detection of the paddle, we could have chosen to use the YCrCb scheme instead of RGB. Changes to the lighting in the environment cause RGB values to all fluctuate. If the room gets bright the R, G, and B values will all increase. This makes it difficult to narrow in on a particular colour using the RGB scheme. In contrast, the YCrCb scheme uses a single signal (Y) for luminance. Thus, if the room gets brighter or less bright only the Y component will change. The actual colour components (Cr and Cb) will be unaffected by the change in room brightness. This solution would potentially be more immune to changes in the room brightness and could allow for more consistent and reliable paddle detection.

To create a smoother (flicker-free) movement of images on the screen, we would have to implement the paddle/ball drawer in hardware.

3 Modular Descriptions

3.1 Video to Memory Module

3.1.1 Overview

The Video to Memory Module is responsible for capturing the video input and storing each video frame into the DDR SDRAM memory through a PLB connection. Figure 3 shows the connections of the Video to Memory Module.

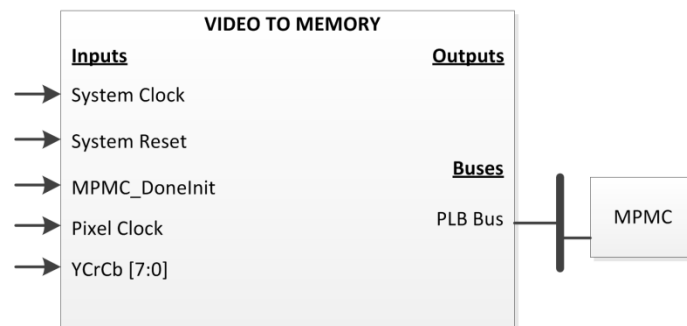


Figure 3: Video to Memory Module

The input to this module is provided by the ADV7183B video processing chip found on the Digilent VDEC-1 daughter board. The chip on the daughter board accepts the analog NTSC video format and converts it to digital values for the Xilinx FPGA. The signals provided by the daughter board are:

- Pixel Clock (1-bit)
- YCrCb video data (8-bit)

The other signals provided to this module are the 100MHz system clock and the system reset. In addition, the MPMC_DoneInit signal is provided. This signal allows the module to wait for the MPMC memory controller to be initialized before requesting writes to the memory.

Figure 4 illustrates an overview of the operation of the Video to Memory Module.

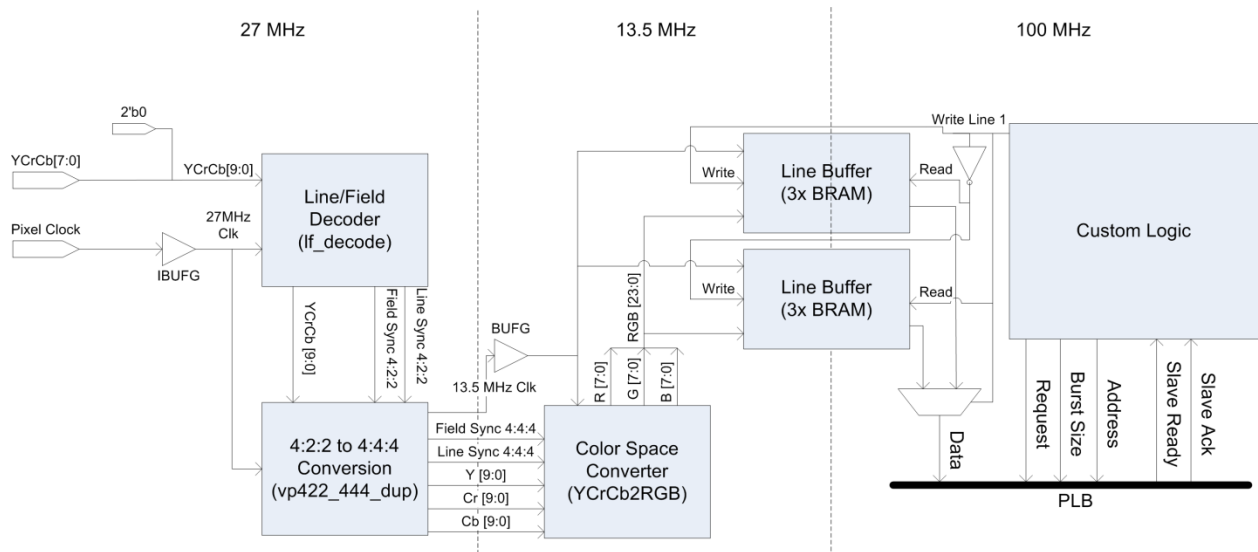


Figure 4: Diagram of Video to Memory Module.

The following sections outline the steps required to process the video input and write the data to the memory.

3.1.2 Line/Field Decoder

The video from the daughter board is provided in the ITU-R BT.656 video format. As with most video formats, the video data is framed using synchronization signals. There are two synchronization signals, field and line. The field signal indicates the beginning of a new frame of video, and the line signal indicates the beginning of a new line of video data. With this video format, the field sync and line sync signals are embedded into the YCrCb video signal using special reserved symbols.

The Line/Field Decoder is required to extract the field sync and line sync signals from the YCrCb video data. The core, *lf_decode*, is provided by Xilinx.

The 8-bit YCrCb signal is routed from the daughter board directly to the *lf_decode* core. The pixel clock is first routed to an input pin clock buffer, *IBUFG*, and then to the *lf_decode* core.

3.1.3 4:2:2 to 4:4:4 Conversion

The input video is provided in 4:2:2 format. This means that the sampling rate of the luminance (Y) component is twice as fast as the sampling rate of the Cr and Cb signals. Before the video can be processed, the sampling rates must be equalized. This is accomplished by converting the video from 4:2:2 format to 4:4:4 format.

Xilinx provides a core, *vp422_444_dup*, which performs the task of equalizing the sampling rates. The field and line sync signals as well as the YCrCb signal are inputted into the core. The core outputs separate Y, Cr and Cb signals as well as adjusted field and line sync signals. The clock is also inputted into the core, and due to the equalization of the sampling rates, the new output clock is divided by 2. Thus, the new clock rate is 13.5MHz.

3.1.4 Colour Space Converter

The next stage of the video processing is to convert the YCrCb video format into RGB video format. The RGB format allows the output of the video to a monitor through a VGA connection and also makes it easier to detect Red, Green and Blue colours.

Xilinx provides a core, *YCrCb2RGB*, which performs the transformation from YCrCb video format to RGB video format.

3.1.5 Dual Line Buffers

Once converted to RGB, the video data is fully prepared to be written to the DDR SDRAM memory. However, the logic to interface with the memory controller is in the 100MHz clock domain. Since data is supplied at 13MHz and written to the RAM at 100MHz, some buffering must occur. A double buffering technique is used with a single line granularity. This means that each line of video is alternately written into one of two available buffers. While one buffer is being filled with video data, the other is being emptied and written into the RAM. With each new line, as indicated by the line sync signal, the operations are swapped. The double buffering technique ensures that there is always an available buffer for incoming video data so that no data is ever lost.

The buffers are being filled at a rate of 13MHz and being emptied at a rate of almost 100MHz. The rate is not quite 100MHz because there is some overhead in requesting memory operations with the DDR SDRAM (See Section 3.1.8). Even with this overhead, the emptying rate is sufficient that the buffer will always be emptied before the next line.

3.1.6 Special SVGA Timing Generation

Another core that is required is the Xilinx *Special SVGA Timing Generation* core. This core accepts the 27MHz pixel clock and the field sync signal. The core produces a line count. This line count is used by the custom logic (3.1.7) to determine what address of the RAM to write the video data to.

3.1.7 Custom Logic

The custom logic is the top-level design and it is essentially the glue around all the modules. It has three main functions:

- Instantiation of Xilinx video modules
- Logic to control double buffers
- PLB Bus master logic

The instantiation of the Xilinx modules includes all modules explained in Section 3.1 and the interconnect between them.

The logic to control the double buffers ensures that each time a new line begins, the read and write signals to the double buffers are swapped.

The PLB Bus master is responsible for the signals required to act as a master on the PLB Bus. Through these signals, the core is able to request burst write operations from the Xilinx Multi-Ported Memory Controller (MPMC).

The timing diagram of the burst transfers are shown in Appendix 5.1.

3.1.8 Bandwidth Calculations

The PLB bus supports a maximum burst size of 16. Since each pixel is word-size, this means that we can successively write 16 pixels to memory before needed to perform another request. Figure 5 illustrates the flow of the PLB master bus requests.

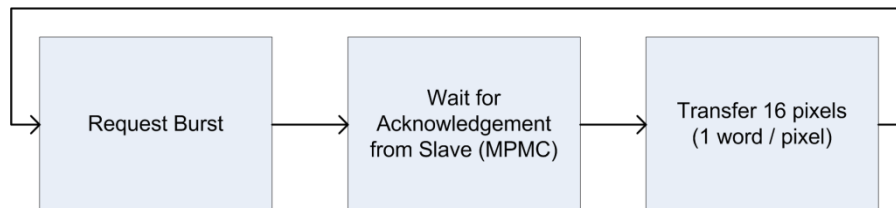


Figure 5: Flow diagram for PLB Burst Write operations required to write the video frame data to the DDR SDRAM.

In order to write a video frame, the above three states are repeated until all pixels in the video frame have been written. The timings of each state are indicated in Table 2.

State	Cycles Required
Request Burst	1

Wait for Acknowledgement	10 (Simulations showed that the memory controller always responded in less than 10 cycles)
Transfer 16 pixels	16

Table 2: State timings of the PLB master write operations.

The maximum number of cycles required to write 16 pixels of video data is:

$$\begin{aligned}
 \text{Max Cycles Required} &= \text{Request} + \text{Wait} + \text{Transfer} \\
 &= 1 + 10 + 16 \\
 &= 27
 \end{aligned}$$

The time required to write all pixels in a line to memory:

$$\begin{aligned}
 \text{Time Per L} &= \frac{\text{Pixels}}{\text{Line}} \div \frac{\text{Pixels}}{\text{Burst}} \times \frac{\text{Max Cycles}}{\text{Burst}} \div \frac{\text{Cycles}}{\text{Second}} \\
 &= 640 \div 16 \times 27 \div 10000000 \\
 &= 10.8\mu\text{s}
 \end{aligned}$$

According to the NTSC video standard, a new line arrives every $63.55\mu\text{s}$. This means that the user logic will always have sufficient time to write the entire line to memory before the next line arrives. The Video to Memory Module is connected to its own PLB Bus; no other masters are present on the bus. This ensures that there will never be contention and the full bandwidth of the PLB Bus will always be available.

3.2 Paddle Detector Module

The Paddle Detector Module is responsible for using the video frame data to detect the endpoints of the two player paddles. Figure 6 shows the connections of the Paddle Detector Module.

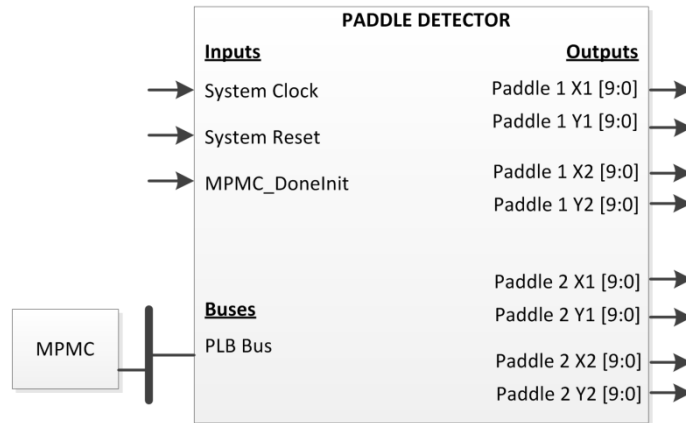


Figure 6: Paddle Detector Module

This module accepts the standard 100MHz system clock and system reset. In addition, the MPMC_DoneInit signal is provided. This signal allows the module to wait for the MPMC memory controller to be initialized before requesting reads from the memory. A PLB connection to the MPMC memory controller allows the module to read the video frame data from memory. The (X, Y) locations of both endpoints of the two paddles are provided as outputs.

Figure 7 shows the flow of operation of the Paddle Detector Module.

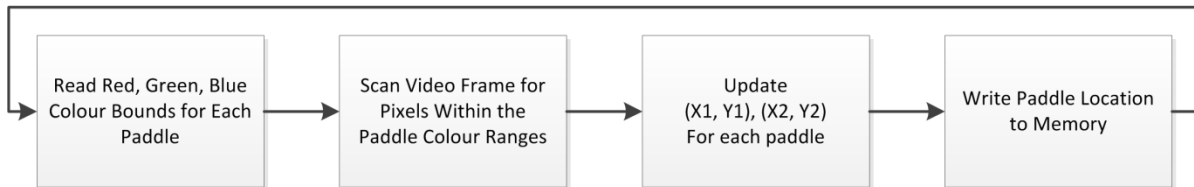


Figure 7: Flow Diagram of the Paddle Detector Module

3.2.1 Setting the Colour Bounds of Paddle Detection

The Paddle Detector Module detects the endpoints of the paddles based on predetermined colours. In order to allow for real-time calibration of the system, these colours are stored in memory that is accessible to the Microblaze processor.

The game is played with two identical paddles, with differing colours at the two ends of the paddle, hereby referred to as End Colour 1 and End Colour 2. We chose to use Red and Green as they could more easily be distinguished using RGB values; however, theoretically any two colours could be used.

Upon initialization of the game, the Microblaze processor writes 12 words of data into memory:

- Lower & Upper Bound of End Colour 1 Red Colour
- Lower & Upper Bound of End Colour 1 Green Colour
- Lower & Upper Bound of End Colour 1 Blue Colour
- Lower & Upper Bound of End Colour 2 Red Colour
- Lower & Upper Bound of End Colour 2 Green Colour
- Lower & Upper Bound of End Colour 2 Blue Colour

These 12 words are repeatedly read by the Paddle Detector Module via the PLB interface. This method was chosen as it allows the Microblaze processor, at any point in time, to change the detection colours. This allows us to adjust to lighting without having to recompile the hardware.

The Paddle Detector core is a master on the PLB Bus, which allows it to request a burst read operation (See Appendix 5.2) to read these 12 words of data. The 12 words of data are stored in registers.

3.2.2 Scanning the Video Frame for Matching Pixels

Once the colour bounds of the paddles have been read, the active video frame can be read from memory. The video frame is read from memory in 16 pixel increments using the same algorithm shown in Figure 5, except that pixel data is read instead of written.

3.2.3 Updating Paddle Coordinates With New Location

As the pixel data is read from the memory, the Red, Green and Blue values of the pixel are analyzed to determine if the pixel corresponds to the end point of a paddle. Figure 8 illustrates the process to determine if the given pixel belongs to a paddle endpoint. If it does, the corresponding (X,Y) coordinate is updated.

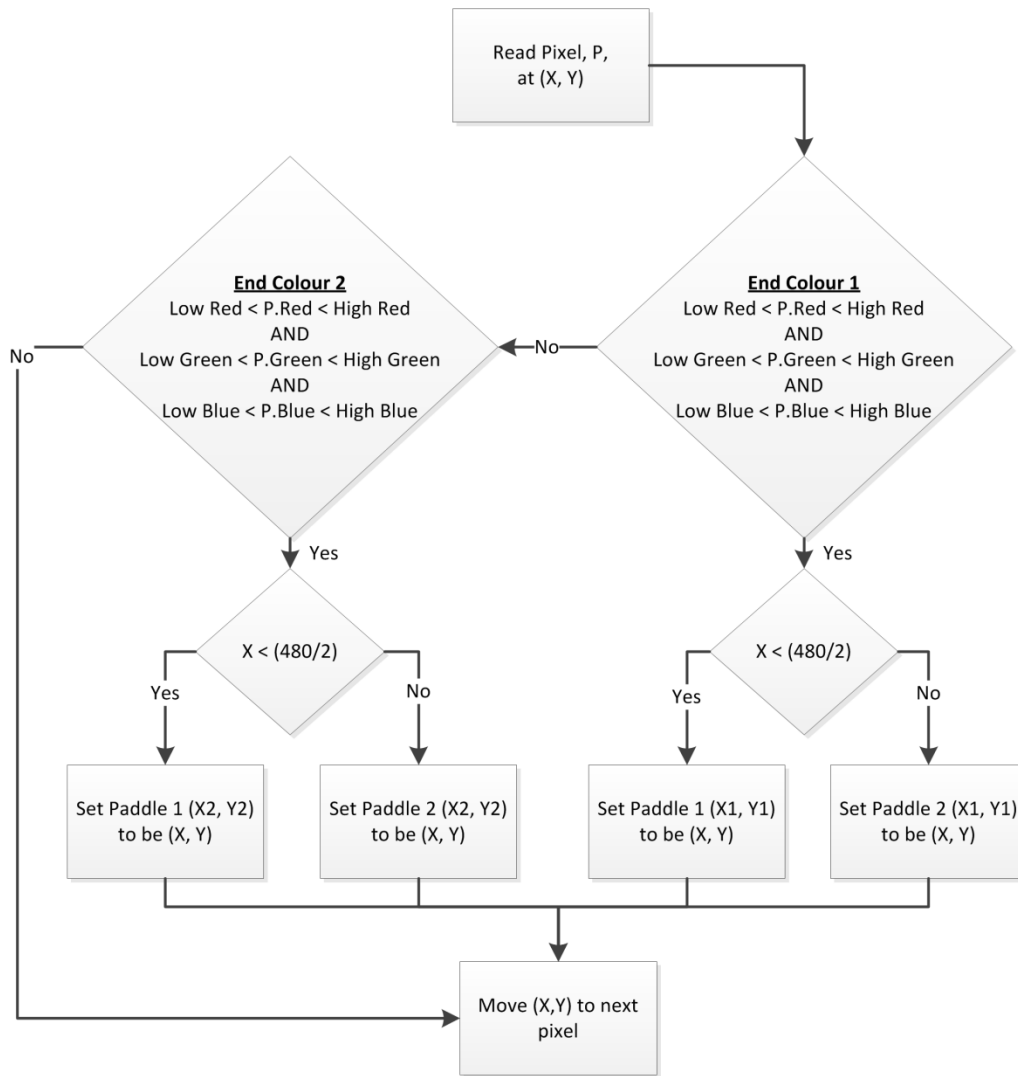


Figure 8: Decision flow to update location of paddle endpoints.

3.2.4 Writing the Paddle Locations to Memory

Finally, the four paddle (X, Y) endpoints are written into memory. The 8 values are written to memory using the PLB interface to the MPMC. The burst write protocol is used (See Appendix 5.1).

These values are written into memory so that they are always available to the Microblaze. This is essential so that the Paddle and Ball Drawer Program (Section 3.3) knows where to draw the paddle.

3.3 Paddle and Ball Drawer Program

The Paddle drawer was created using the Bresenham's line drawing algorithm. The ball drawer was created using a modification of the Midpoint circle algorithm. Both the algorithms were implemented on the MicroBlaze soft core processor.

3.3.1 Paddle Drawer

The four points of the two paddles were detected in the Paddle detector module and output to the processor using the XPS General Purpose IO IP provided by Xilinx. The port connections between the Paddle Detector and the Gpio were made using the EDK GUI and were made available through the program using `XGpio_DiscreteRead(XGpio *InstancePtr, unsigned Channel);` Each XGpio IP had two channels corresponding to the two points for each paddle. Using the Bresenham's line drawing algorithm, a line was drawn to connect the two points.

The line is erased (drawn black) and re-drawn at each iteration of the while loop.

3.3.1.1 Paddle Drawer Algorithm

A standard Bresenham's line drawing algorithm was used. It works as follows:

- Check if the line is a steep line (i.e. change in Y is bigger than the change in X), in which case, swap the X and Y co-ordinates of both points.
- Make sure that X1 is smaller than X2, if this is not the case, do the necessary swapping of the points.
- If Y1 is smaller than Y2, then each step (Ystep) must be a negative value. Otherwise, it must be positive. In our case, we are incrementing by 1.
- Loop from X1 to X2, if the line is steep, plot (Y,X) as X,Y on the screen, otherwise, plot (X,Y) on the screen
- If the next point requires a Y-increment, then perform the corresponding Ystep increment. This is checked using an error variable which is initialized to dx/2 (half of the change in X), and after each plot, dy is subtracted from it. If the result is negative,

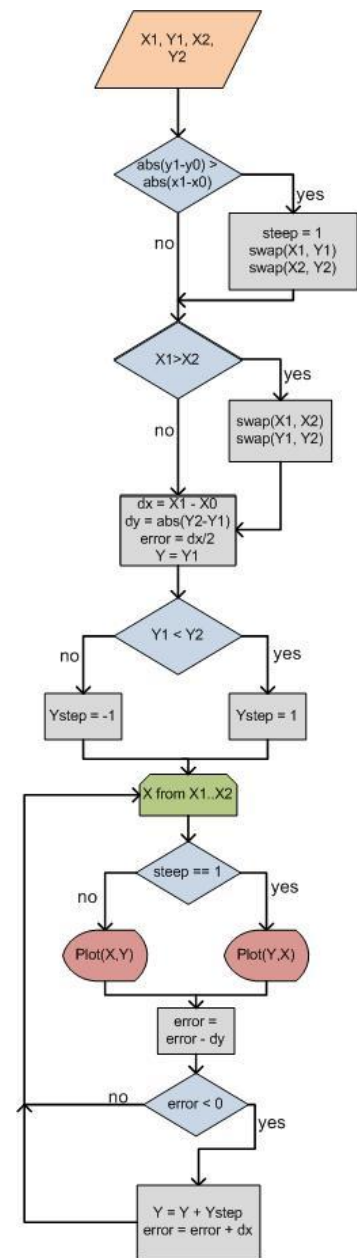


Figure 9: Bresenham's Line Drawing Algorithm

then an increment is necessary. See the Algorithm diagram for an illustration.

3.3.2 Ball Drawer

The location of the ball is calculated by **ball_control** core (See 3.3) and the X, Y co-ordinate is supplied to MicroBlaze through the XGpio interface. The co-ordinate is updated at a configurable number of clock cycles depending on the velocity of the ball. This value is read by the software and the ball is drawn.

The ball is erased (i.e. drawn black) and redrawn at each iteration of the while loop.

3.3.2.1 Ball Drawing Algorithm

A standard **Midpoint Circle Drawing Algorithm** was used as explained in Wikipedia [1]. However, this algorithm does not fill in the circle. Therefore, the algorithm was modified such that instead of plotting two pixels, a line was drawn between them. This filled in the circle with a solid colour.

3.3.3 Functions used in MicroBlaze

```
//Create and XGpio variable
```

```
XGpio Instance;
```

```
//Initialize the variable with the device ID as generated in xparameters
```

```
intXGpio_Initialize(XGpio *InstancePtr, u16 DeviceId);
```

```
//Read from the particular GPIO port – corresponds to a memory-mapped port as connected in the
```

```
//system diagram
```

```
u32 XGpio_DiscreteRead(XGpio *InstancePtr, unsigned Channel);
```

```
//Write to a particular GPIO port – corresponds to the port as connected in the system diagram
```

```
u32 XGpio_DiscreteWrite(XGpio *InstancePtr, unsigned Channel,u32 Mask);
```

3.4 Sound Module

The goal of this module was to produce sound each time the ball interacts with the paddle, or to indicate the start or end of a game. There are four possible sounds that can be produced: A single toned beep during a ball-paddle collision, a double toned beep when a player scores, a 5-toned sound when a new game is started and a 6-toned sound when the game ends. A third party verilog to interact with the AC'97 LM4550 chip on the XUP-V2P board was found in [2]. The HDL was able to direct the audio input from a mic to the speakers. It also produced a square wave that was output to the speakers. This was modified to produce square waves of different frequencies to correspond to the different tones. An 11 bit counter was used for this purpose, where each bit of the counter produced a square wave of different frequency (e.g. *counter[0]* produces a square wave that is double in frequency from *counter[1]*).

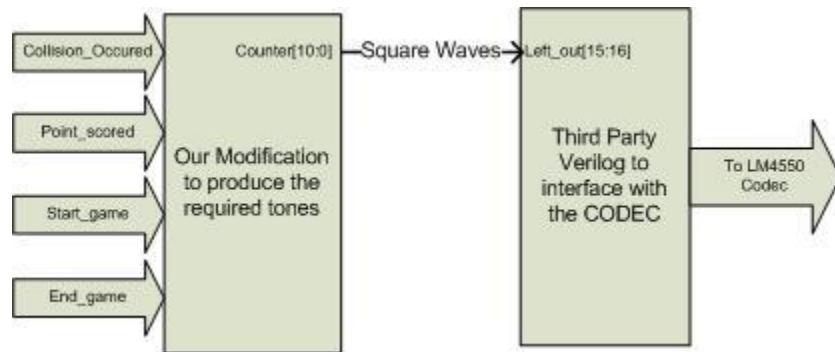
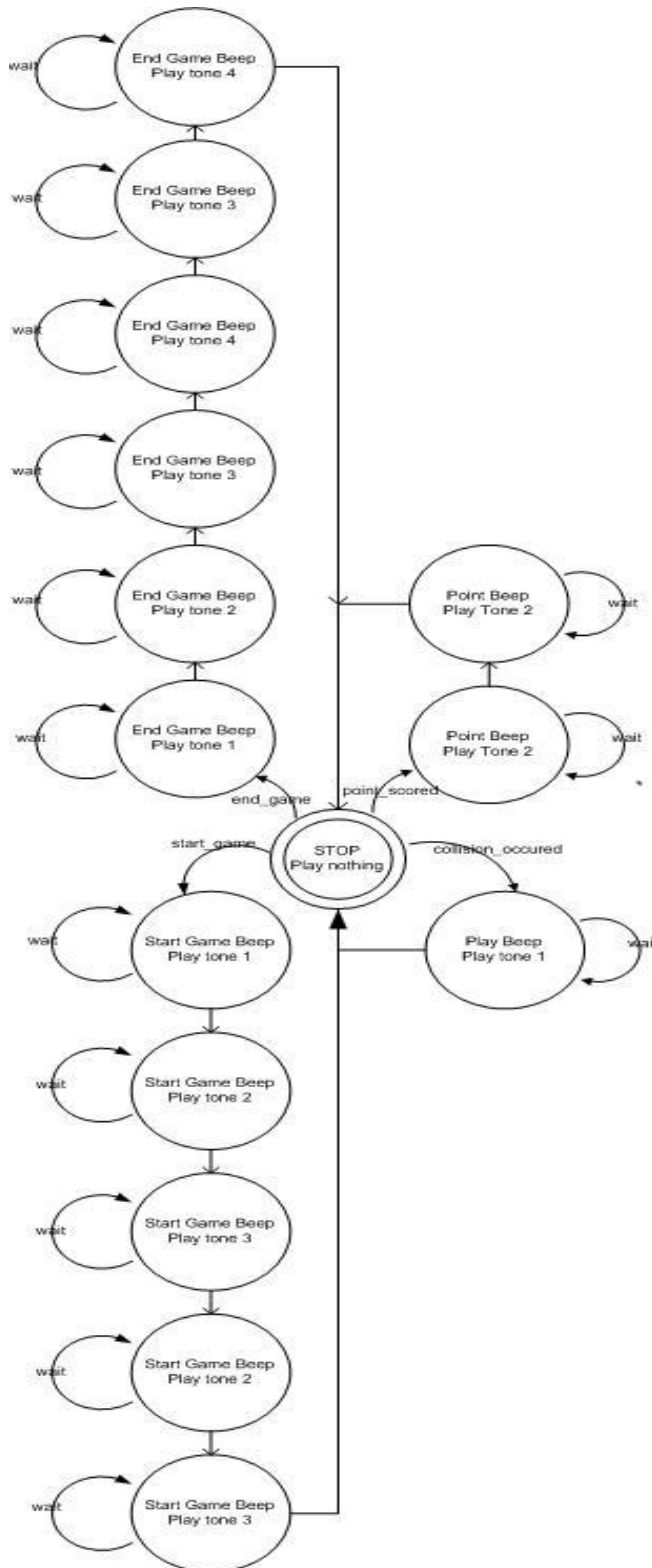


Figure 10: Sound Module interactions

Collision_Occured, **Point_scored** is produced by the **ball_control** core. The **Start_game** and **End_game** signals are produced by the program in MicroBlaze and interfaced through the XGpio interface.

3.4.1 FSM to produce different tones



This FSM produces 4 distinct sounds. The Beep state is triggered with the **collision_occured** signal is a single state where we wait a certain amount of time, during which a tone is output to the speakers. Similarly, the End Game Beep State is triggered by the **end_game** signal, the Start Game Beep state by the **start_game** and the Point beep by the **point_scored** signal. This is a moore-type state machine where the output is controlled based on the current state.

To produce a multi-toned sound, multiple states are cycled through and a different tone is played depending on the state. To change the tone (i.e. frequency of the square wave), a different bit of an 11-bit counter is assigned the speaker. The tones in the state machine diagram shown correspond to:

Tone 1: counter[10]

Tone 2: counter[9]

Tone 3: counter[8]

Tone 4: counter[7]

Figure 11: State Diagram for Custom Logic

3.4.2 Sound Module Diagram

The Audio module interfaces with the clock generator, the Ball Control custom module, MicroBlaze, as well as external ports.

Clock generator supplies a buffered clock, the Ball Control module identifies the points of collision and when a player scores a point. The corresponding values are held high for one clock cycle, and are read by the audio core custom logic. The 'Start Game' and 'End Game' signals are supplied by the MicroBlaze program through the XGpio interface. The Audio module has external pins, which connect to the LM4550 chip in the XUP-V2P board. A modular diagram is presented below:

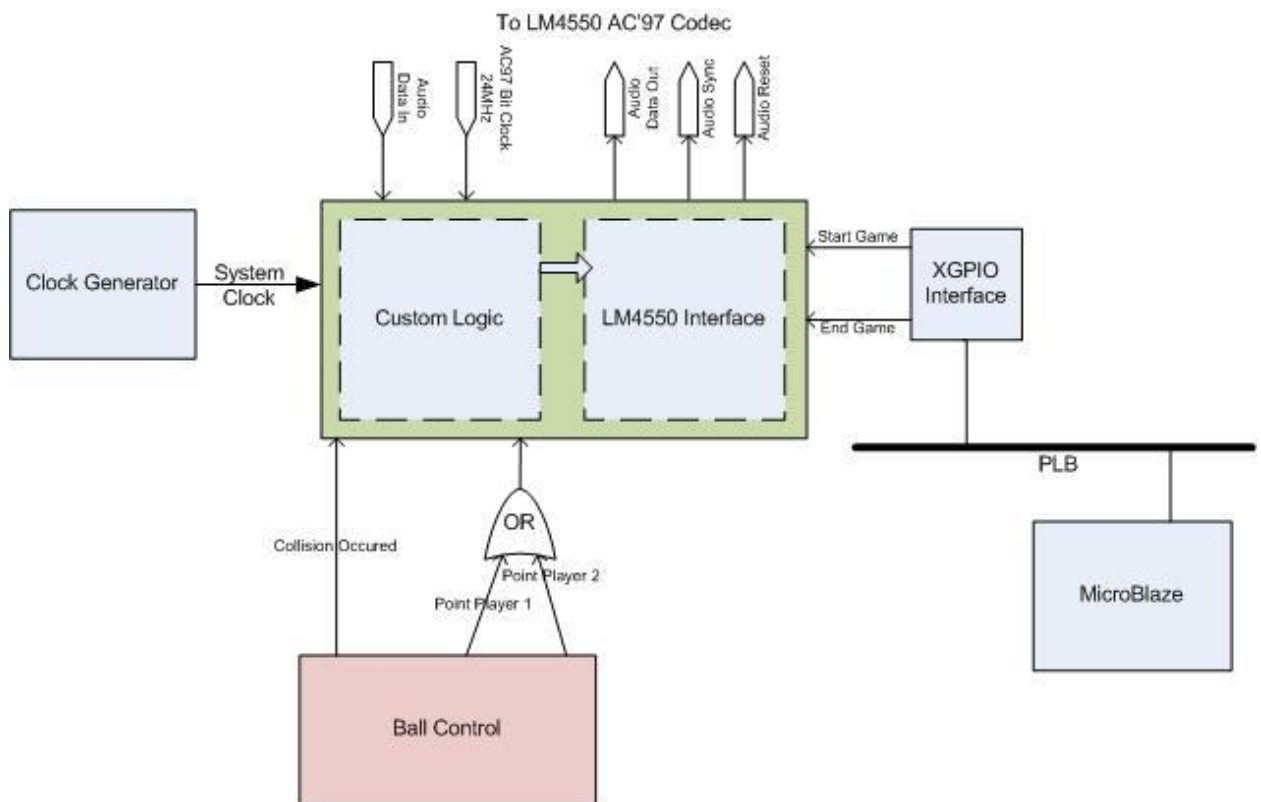


Figure 12: Sound Module Diagram

3.5 Ball and Score Control Module

The Ball Control Module is responsible for the physics of the project. It is given the detected points from the Paddle Detector Module. It utilizes this information to keep track of the position of the ball. Upon a collision of the paddle with the ball, it will change the velocity of the ball and as a result, the position of the ball will be updated with the new velocity. After it completes this, it sends the ball co-ordinates to the “Paddle and Ball Drawer Program” on the Micro blaze which will draw it on the screen periodically.

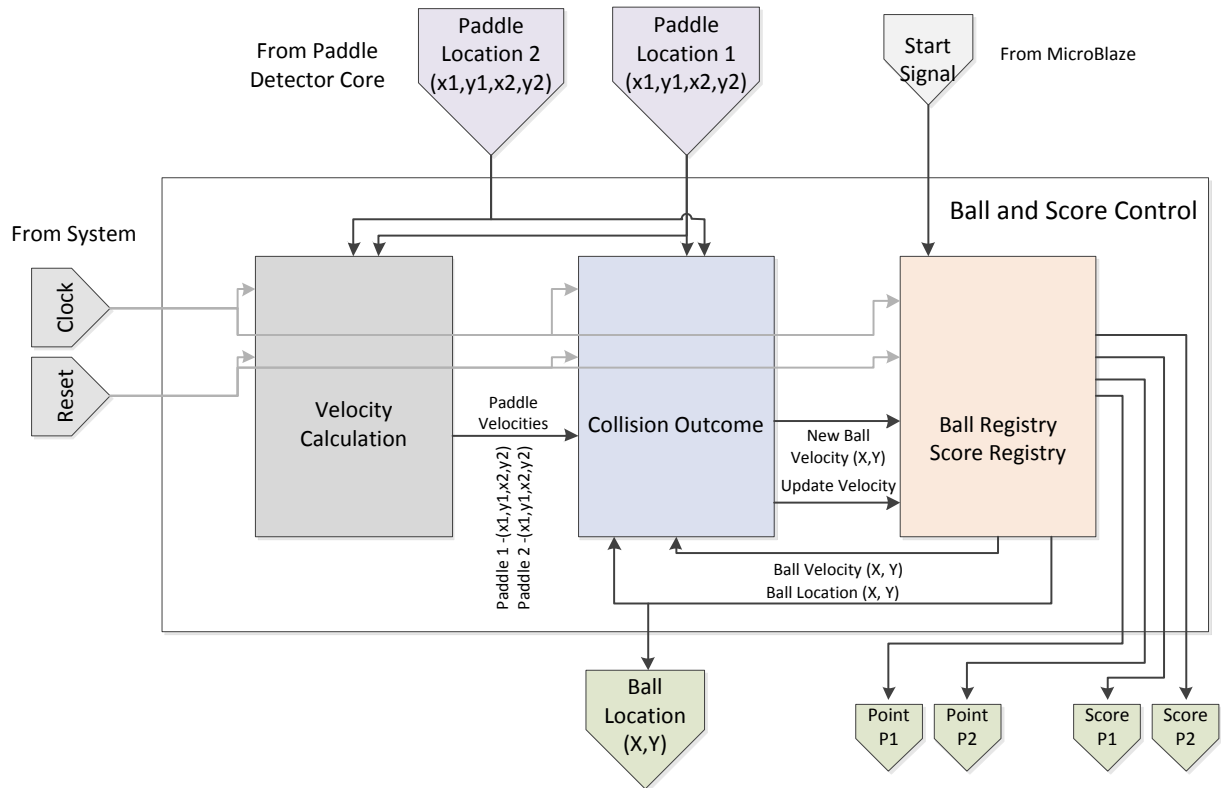


Figure 13 Ball and Score Control Complete Module

3.5.1 Velocity Calculator

This core determines the velocity of the Paddle. This velocity is required in order to determine the speed of collision between the ball and the paddle.

Sources of Input (Refer to I/O Diagram below)

- Paddle Detector Core – Locations of Paddles
- Sampling Rate Parameter – Made configurable through EDK

Functions:

- Register the input velocity
- Periodically (Based on Sampling Parameter) determine the difference between the old velocity and the new velocity
- Determine the new velocity vector
- Ensure the velocity does not go out of bounds (parameter Max Value)

IO Diagram & State Diagram:

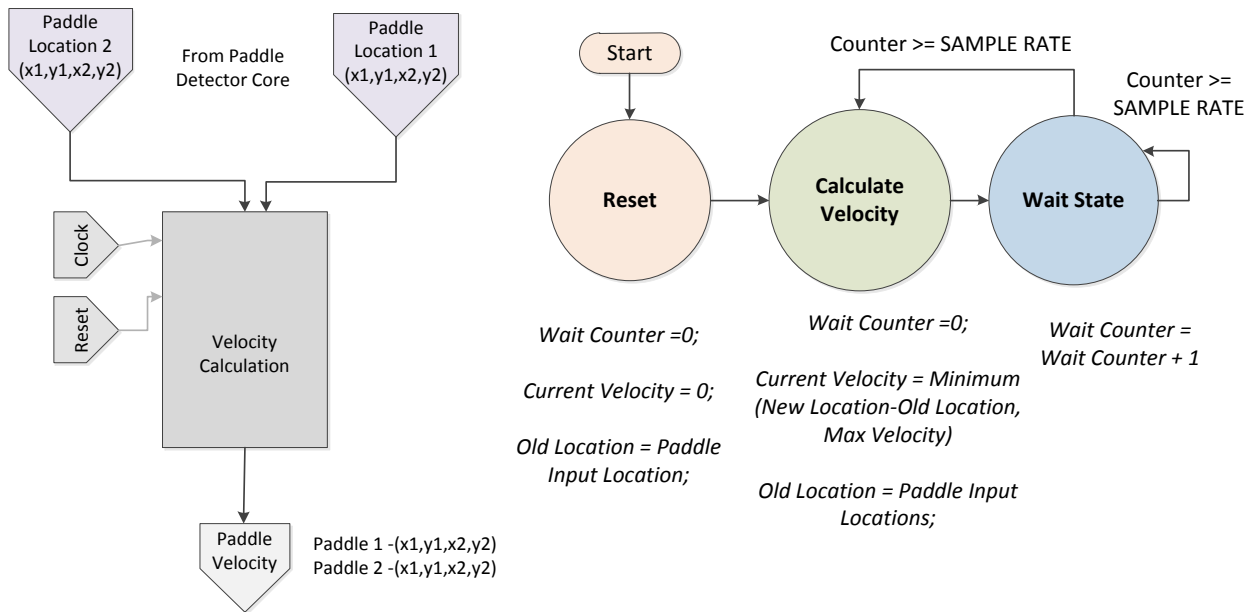


Figure 14 Velocity Calculation Sub Module IO Diagram and State Machine

3.5.2 Collision Detector

Given the position and speed of the Paddle as well as the ball's co-ordinates + velocity, we can determine when the objects collide and the resulting velocity of the ball.

Sources of Input (Refer to I/O Diagram below)

- Paddle Detector Core – Locations of Paddles
- Velocity Calculator - Velocity of each Paddle point
- Ball Registry – Current Ball Location and Velocity

Functions

- Determine when a collision occurs based on the 2 points of each paddle and the ball location.
- Determine the resulting velocity of the ball
- Determine when the Ball Registry should update its Value

I/O Diagram

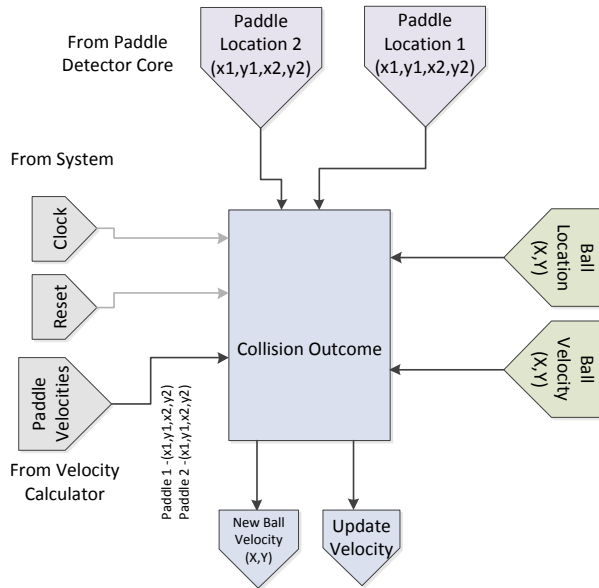


Figure 15 Collision Outcome I/O Diagram

State Diagram

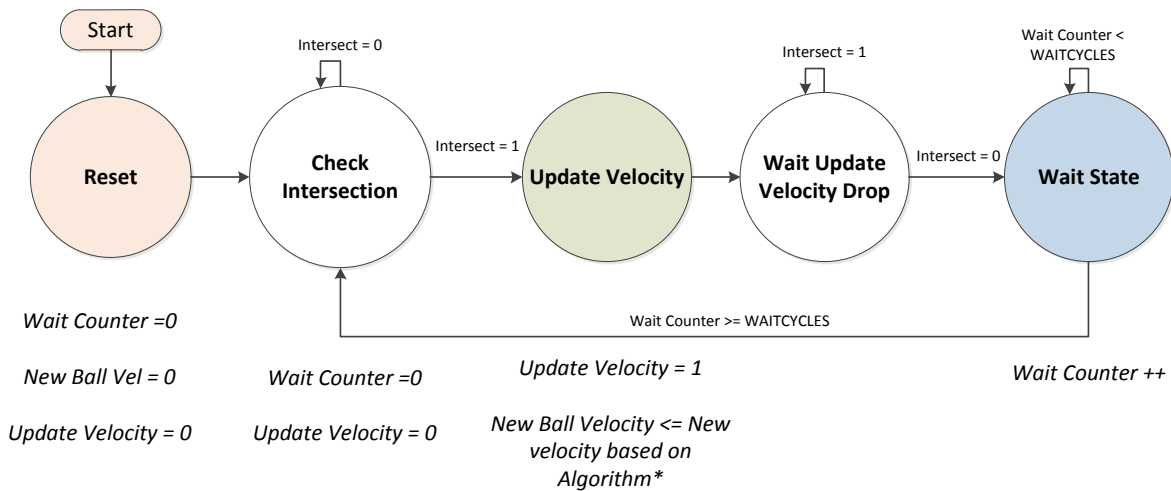


Figure 16 Collision Outcome State Machine

***Resulting Velocity Calculation Measurement**(NOTE: We treat the X and Y directions independently)

CASE1: BALL AND PADDLE MOVING IN THE SAME DIRECTION

if ((Current Ball Velocity is positive AND Velocity of Paddle is positive) OR
(Current Ball Velocity is negative AND Velocity of Paddle is negative))

$NewBallVel = CurrentBallVel + VelPaddle;$

CASE2: BALL AND PADDLE MOVING OPPOSITE DIRECTIONS

else if ((Current Ball Velocity is positive AND Velocity of Paddle is negative) OR
(Current Ball Velocity is negative AND Velocity of Paddle is positive))

$NewBallVel = -CurrentBallVel + VelPaddle;$

CASE 3: ALL OTHER CASES:

else

$NewBallVel = -CurrentBallVel;$

Determining when a collision has occurred

We use mathematics to determine if a collision has occurred. We base our decision on 2 main conditions:

- **CONDITION 1:** The intersecting ball should fall within the regions contained within the box of the paddle co-ordinates. For example:
 - $X1 < X < X2$ OR $X2 < X < X1$
 - $Y1 < Y < Y2$ OR $Y2 < Y < Y1$ where $X1, X2, Y1, Y2$ are the paddle points.
- **CONDITION 2:** The intersecting ball will lie along the line and will contain the same slope as the line of intersection. Given Paddle Point 1 ($X1, Y1$) and Paddle Point 2 ($X2, Y2$) and Ball point ($X3, Y3$) we can compare the slopes with the following equation.
 - Slope = Difference in Y Co-ordinates/Difference in X Co-ordinates
 - $(Y3 - Y1) / (X3 - X1) ==? (Y2 - Y1) / (X2 - X1)$
 - Cross Multiplying (To avoid dividers) we get:
 - **$(Y3 - Y1) * (X2 - X1) ==? (Y2 - Y1) * (X3 - X1)$**

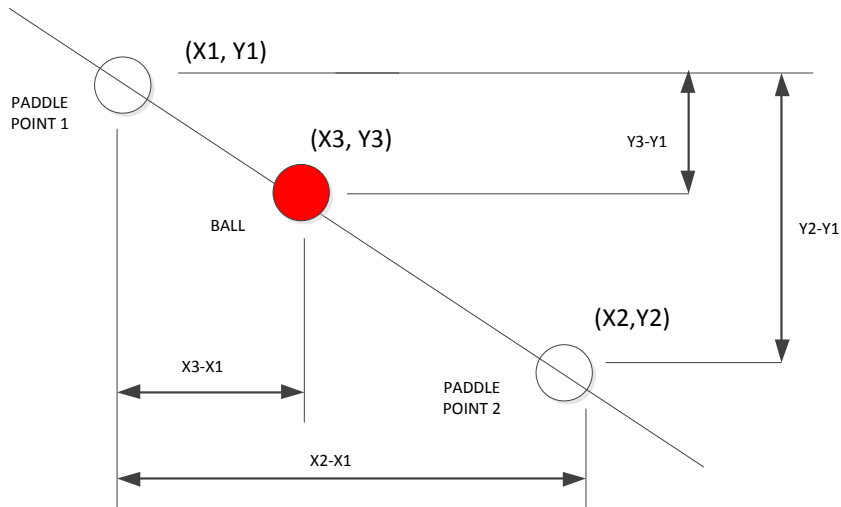


Figure 17 Collision Outcome Condition 2 Slope Depiction.

3.5.3 Ball Registry and Score Registry

The ball registry is responsible for keeping track of the location and velocity of the ball. As the game progresses, it periodically updates the ball's position on the screen.

The score registry keeps track of the score based on the ball collisions with the left and right of the screen. If a collision is detected, then the score is incremented and a signal is sent to the Audio core so that a sound is played.

Sources of Input(Refer to I/O Diagram below)

- Collision Outcome - New calculated Velocity and Update Velocity Signal
- MicroBlaze – Start Game Signal

Functions

- Store the current ball location and velocity
- Update the ball location based on the velocity
- Allow the velocity to be altered based on Collision Outcome core.
- Determine if the ball goes out of bounds and if so, switch the velocity so it stays within bounds
- Notify the Audio core when a sound should be played (collision with walls or point for a player)
- Keep track of the score of players (collisions with Left and Right side)

I/O Diagram

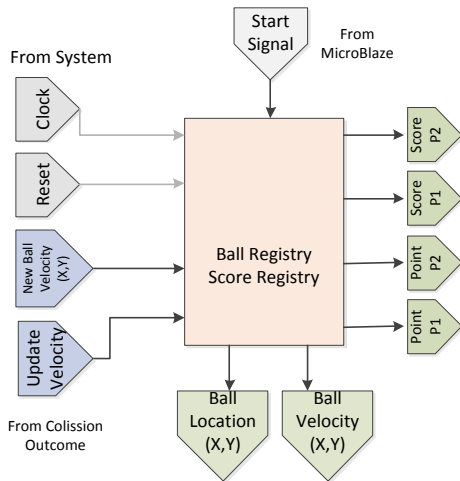


Figure 18 Ball and Score Registry I/O Diagram

State Diagram

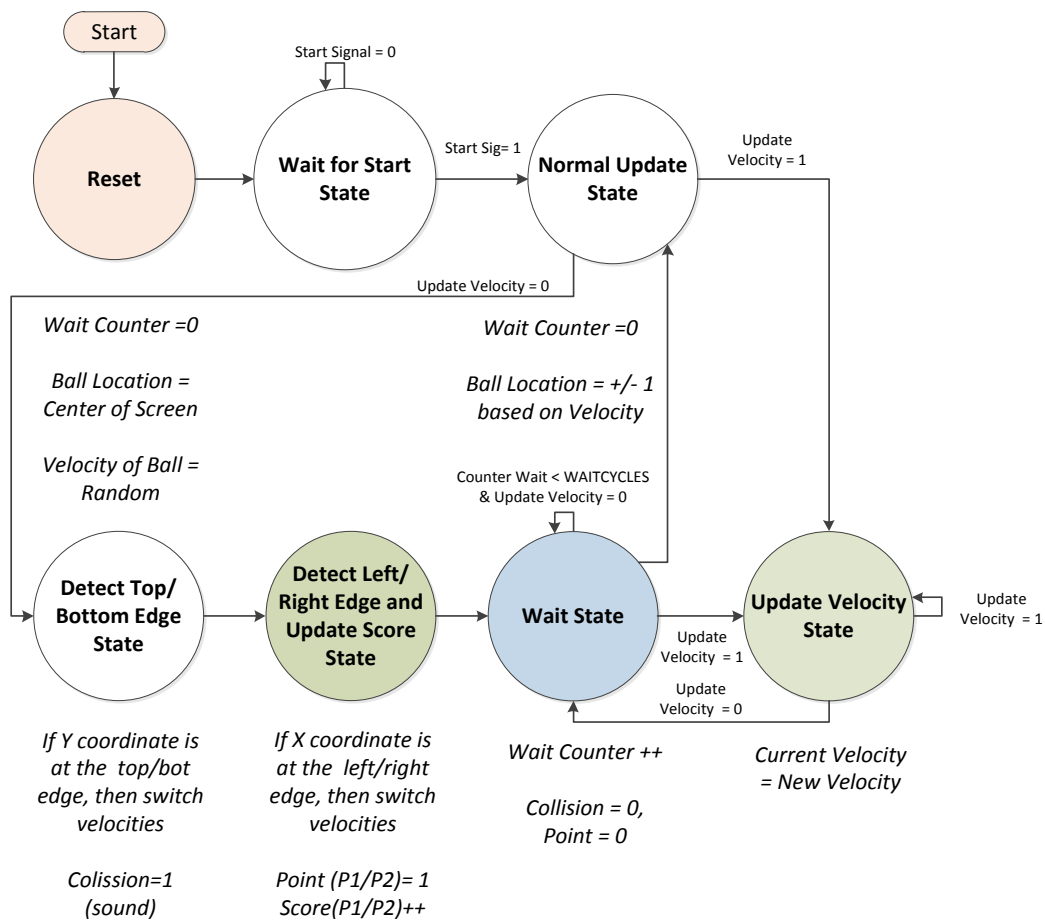


Figure 19: Ball and Score Registry State diagram

Update Position based on Velocity

When updating the balls position, we considered two alternatives for updating the ball.

- Alternative 1: Update position every fixed period (Parameter) by Velocity (0, 1,2, or 3) pixels
- Alternative 2: Update position by 1 unit per period; however the period will vary with Velocity.

We initially chose Alternative1, however the game play was not smooth to our liking and velocities could only increase in multiples of the original velocity. In Alternative 2, we have implemented a configurable period where we can determine the frequency of updates based on the velocity. The Implementation involved splitting the FSM shown above into two each having independent Wait Counters. By alternating the comparison value of the counter, we can implement different speeds.

Detection of Edges

We detect if the ball is positioned near the edges of the screen. The purpose is to prevent the ball from going out of bounds, keep track of the score, and send a signal to the sound control to output a sound when a collision with walls are detected.

In the current implementation, the ball has a radius of 10. The ball's location is also determined by the center of the ball. To prevent the ball from appearing outside the bounds of the screen, we ensure that the center of the ball stays within R (10 pixels) of the edges at all times. If we detect a ball is within the edge region, we reverse the velocity direction to bounce the ball back.

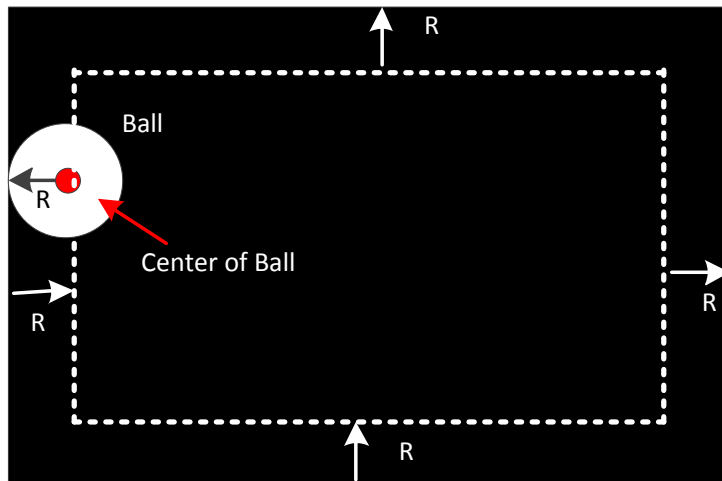


Figure 20 Edge Detection of Ball on Screen

3.5.4 Testing the Ball and Score Control Module

In order to test this module prior to integration, ISE and ModelSim was used in conjunction. ISE was used to develop the HDL and ModelSim was used in all stages to verify that the code was synthesizable and that it worked as expected before and after synthesis (Translation).

Each Sub-Module (Velocity Calculator, Collision Outcome, and Ball/Score Control) were independently developed and tested (before and after synthesis). In addition, a useful feature we utilized in ISE was the test bench skeleton generator as well as the Core Generator which was used to integrate a divider into the calculation for collisions.

Below is a ModelSim screenshot showing a moving ball hitting a stationary paddle. The outcome of the collision switches velocity.



Figure 21 ModelSim simulation. Basic case depicting a collision of a stationary paddle with a moving ball

4 References

[1] Midpoint Circle Algorithm, [Online] <http://en.wikipedia.org/wiki/Midpoint_circle_algorithm>

Accessed: March 10, 2010

[2] AC'97 Audio Codec controller for Digilent XUP-V2P, [Online]

<http://embedded.olin.edu/xilinx_docs/projects/audio-v2p.php> Accessed: March 17, 2010

5 Appendix

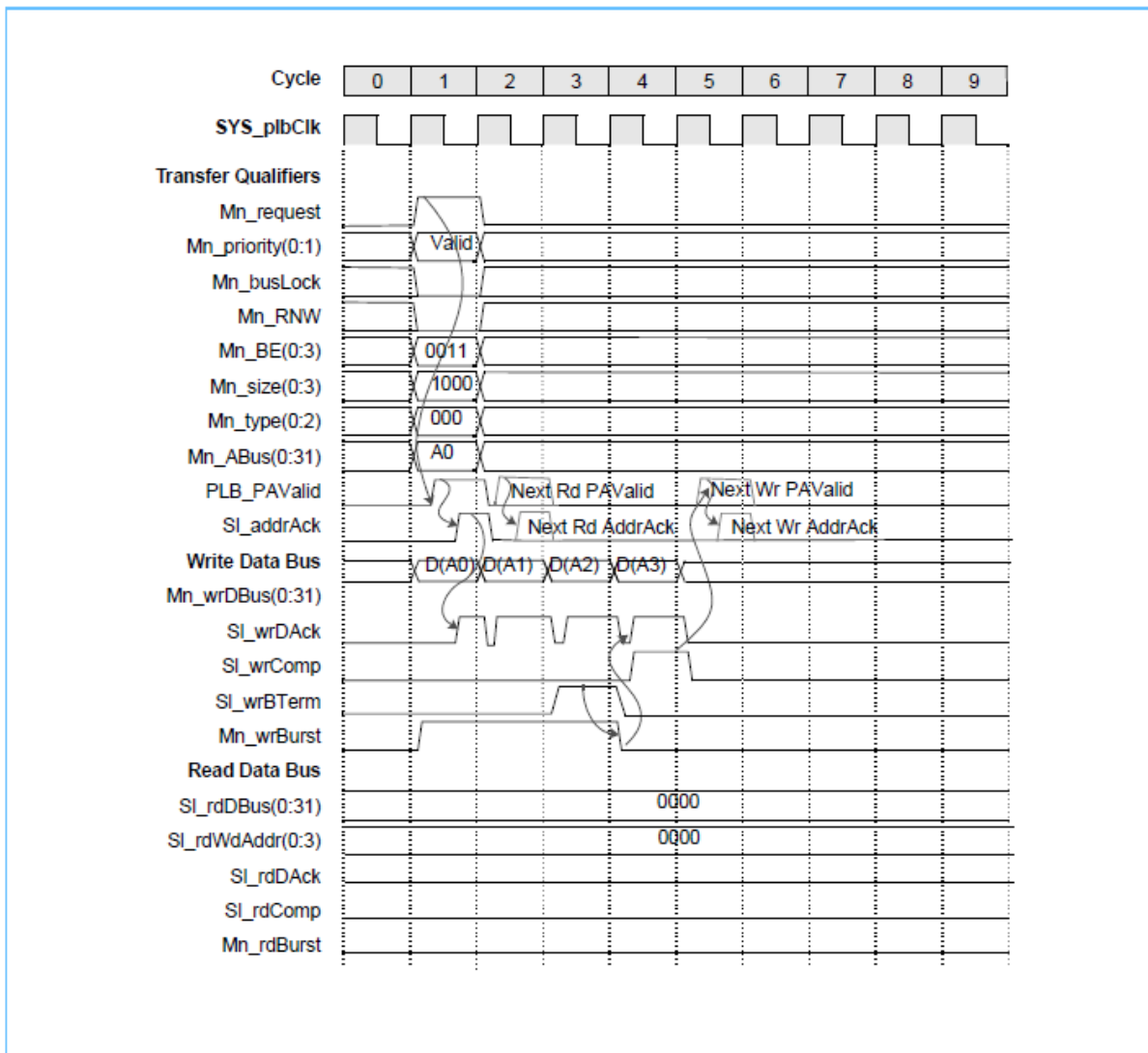
5.1 PLB Master, Burst Write Operation



5.1.16 Fixed-Length Burst Write Transfer

Figure 5-15 shows the operation of a fixed-length burst write from a slave device on the PLB. During the request phase of the transfer, the master has continuously provided the length of the burst on the Mn_BE signals and is requesting to write 4 words. The slave uses this length value to count the number of transfers and assert the SI_wrBTerm signal in the cycle before the last assertion of the SI_wrDack signal.

Figure 5-15. Fixed-Length Burst Write Transfer



5.2 PLB Master, Burst Read Operation



Architecture Specifications

128-Bit Processor Local Bus

5.1.15 Fixed-Length Burst Read Transfer

Figure 5-14 shows the operation of the fixed-length burst read from a slave device on the PLB. During the request phase of this transfer, the master has optionally provided the length of the burst on the Mn_BE signals and is requesting to read four words. The slave uses this length value to count the number of transfers and assert SI_rdBTerm in the cycle before the last assertion of the SI_rdDack signal. This allows a subsequent read transfer to be acknowledged.

Figure 5-14. Fixed-Length Burst Read Transfer

