ShadowBall

6.111 Final Project

Fall 2001

Sourav Dey | Manu Seth | Sean Lie

Abstract

A human video interaction system was designed and implemented. The goal was to achieve a high level of human interaction with a virtual object. Using a video camera and a monitor the user of this system is able to “touch” a virtual ball and interact with it through his physical motion. In addition, the goal was to design a physics model that would produce realistic on-screen ball motion while being simple to implement and expand if necessary.

This project is dedicated to Vikram Maheshri, without whose brilliant wisdom we would have been struggling with a glaucoma tester.

Table Of Contents

1. Overview 5
2. Description 7
    A. Video Processing Unit 7
        i. Video Processing 8
        ii. NTSC Video 8
        iii. Sync Separator 9
        iv. Analog to Digital Converter 9
        v. Pixel Counters 10
        vi. Digital to Analog Converter 11
    B. Storage Unit 12
        i. Control FSM 13
        ii. RAM Addressing 14
        iii. Buffer Swapping 15
        iv. Data I/O 15
    C. Ball Processing Unit 16
        i. Physics Algorithm 16
        ii. Control MCU 19
        iii. State Registers 21
        iv. Ball PROM 21
        v. Quadrant Intersection Detection Logic 24
        vi. Physics Lookup PROM 26
        vii. Update Logic 28
3. Testing and Debugging 29
    A. Ball Processing Unit 29
        i. Moving Through Solid Objects 29
        ii. Horizontal Jumping 30
        iii. Random Movement 30
        iv. Downward Drift 31
    B. VPU and Storage Unit 32
4. Conclusion 32
5. Appendix 34
    A. Detailed Storage Unit Diagram 34
    B. Detailed BPU Diagram 35
    C. VPU VHDL Code 36
    D. Storage Unit VHDL Code 43
    E. BPU VHDL Code 48
    F. BPU Microcode 56

List of Figures

1. ShadowBall Example Screenshot 5
2. Overall System Block Diagram 6
3. VPU Block Diagram 7
4. NTSC Signal Format 9
5. Pixel Counters 10
6. Storage Unit Control FSM 13
7. Ball Quadrant Encoding 17
8. Ball Direction Encoding 17
9. BPU Control MCU 19
10. BPU Block Diagram 20
11. Ball PROM Pixel Location 22
12. Ball PROM Data Format 22
13. Ball Pixel Address Logic 24
14. Quadrant Intersection Detection Logic 25
15. Physics Lookup PROM Excerpt 27

Overview

The overall system will be comprised of a video camera and a video monitor. The video camera will be positioned such that the hands of the user are in its frame of view. On the monitor, the user will see the real-time video of the input camera. Therefore, when the user moves his hands, this movement will be displayed on the monitor.

Superimposed on top of the real video will be a picture of a ball that is added by the system. This ball will be moving on the monitor and the user will be able to interact with it by moving his hands. The system will treat all white pixels on the monitor as being solid. The background color behind the user's hands will be black making the hands will appear light on the video monitor. When the ball on the monitor comes into contact with the picture of the user's hand, it will bounce off of it as if it were a solid object. In this manor, the user can interact with the virtual ball on the monitor by moving his hands and "touching" it.

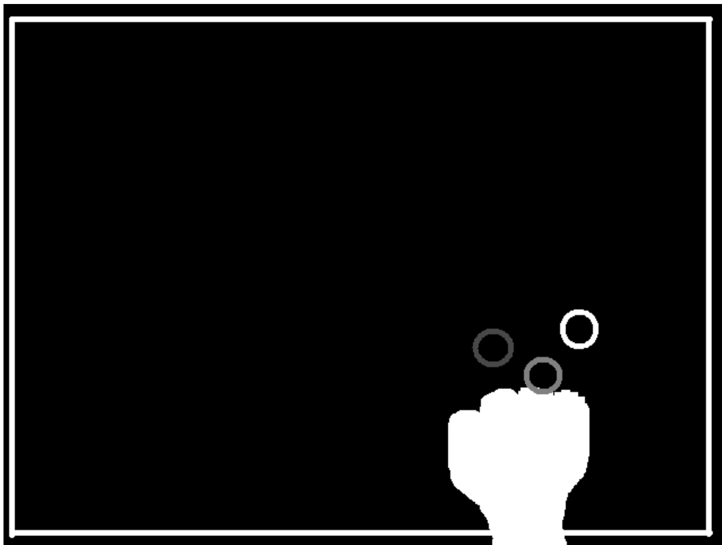


Figure 1: ShadowBall Example Screenshot

The ShadowBall system is divided into three components: the Video Processing Unit (VPU), the Ball Processing Unit (BPU), and the Storage Unit. The VPU is responsible for processing the input from the video camera and the output to the monitor. Figure 2 shows an overall block diagram of the system. The Storage Unit contains an input and an output buffer. The VPU writes camera data into the input buffer while reading data from the output buffer to be displayed on the monitor. Once one frame has been simultaneously read from the camera and written to the monitor, control of the Storage Unit is given to the BPU. In between the frames, the BPU adds the image of the ball to the input buffer in the Storage Unit. After the BPU has added ball, the two buffers in the Storage Unit are switched and the next frame is read from the camera and written to the monitor.

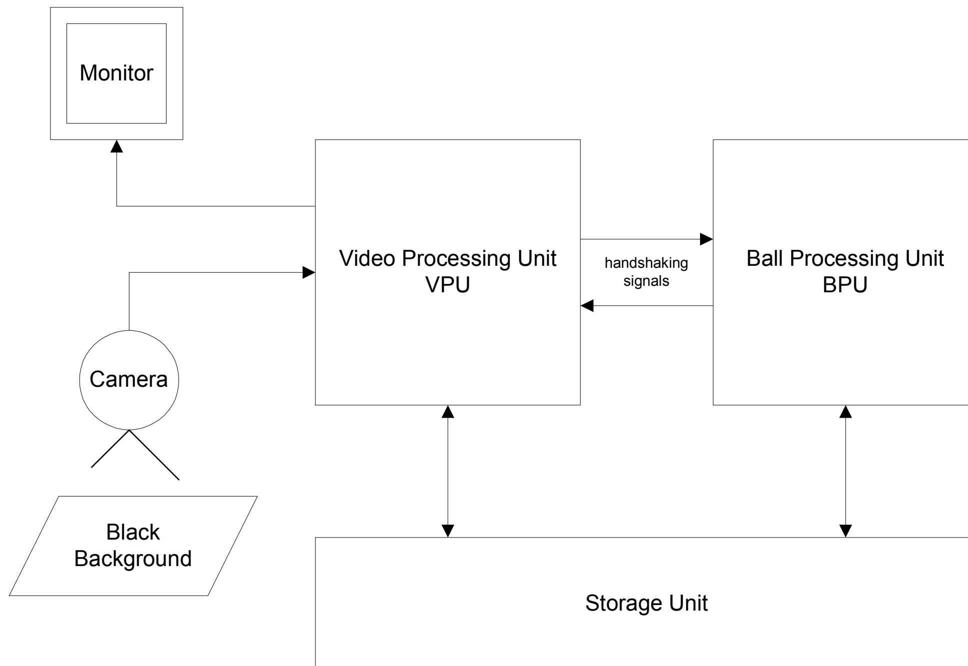


Figure 2: Overall System Block Diagram

Description

Video Processing Unit

The video processing and memory unit takes data from the camera, stores it in RAM, and then outputs the image with the ball to the screen. This can be divided up into video functions and storage functions.

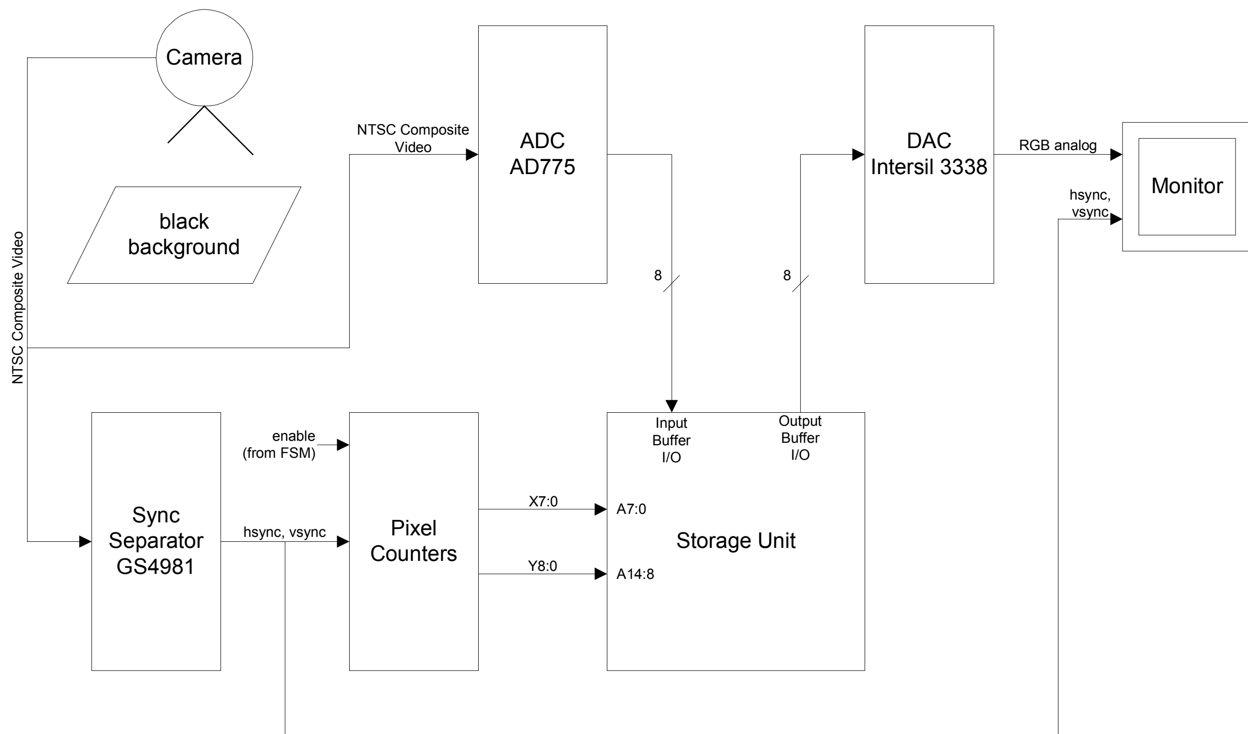


Figure 3: VPU Block Diagram

Video Processing

Shadowball uses an NTSC camera with a composite video output. A GS4981 Sync Separator chip is used to extract the hysnc and vsync signals from the composite signal. The hsync and vsync are used to drive the monitor synchronously with the camera. They are also used for addressing the RAM and determining when to process the image. The composite output also goes to an AD775 high speed Analog to Digital Converter. The 8 output bits are sent to the storage unit. The storage unit outputs the image as an 8 bit signal from the RAM. This signal is sent through an Intersil 3338 flash Digital to Analog Converter. The analog output is used to drive the red, green, and blue channels on a color monitor, creating a 256 shade grayscale image with a resolution of 128x256.

The VPU works with the storage unit to simultaneously store one frame, while outputting another frame of video. The time between frames of video is used to process the ball and to swap the input and output RAMs.

NTSC Video

The video camera outputs a composite video signal in the NTSC standard. NTSC video consists of analog video that is output for each horizontal scan line. At the end of each scan line there is an hsync signal during which time the electron beam retraces to the start of the ext line. Figure 4 shows the format of an NTSC signal. The camera outputs 256 lines of video per frame. After these 256 lines have been outputted, there is a vertical sync signal and vertical blanking

period during which the electron beam retraces to the top left of the screen. The vertical blanking time is significantly longer than horizontal blanking (~ 7 us). Shadowball uses this vertical blanking time to do all its video processing for the ball.

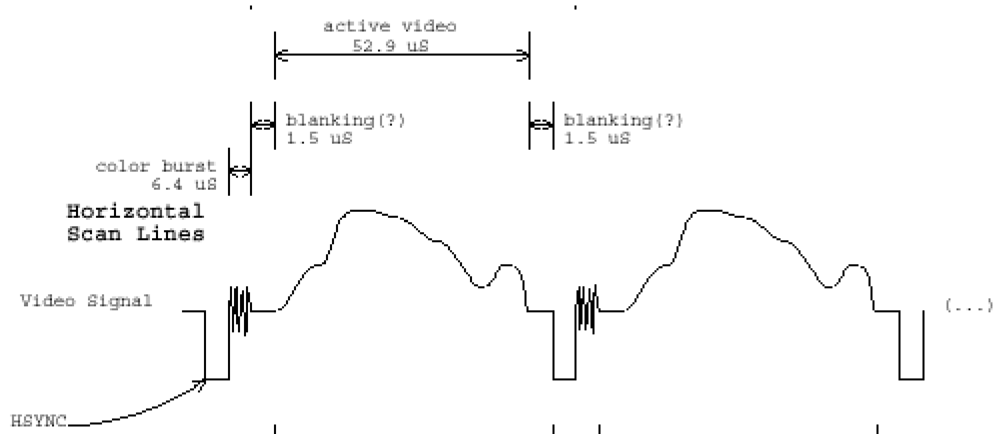


Figure 4: NTSC Signal Format

Sync Separator

The GS4981 sync separator chip takes in the analog video signal and outputs a digital hsync and vsync signal. These sync signals are used as control signals to the FSM, as well as enables for the x and y pixel counters.

Analog to Digital Converter

An AD775 analog to digital converter is used to digitize the video signal. This high speed A2D converter is able to digitize at video speeds. It requires no input signals aside from a sampling clock. The system uses a 5 MHz clock and this is also used as a sampling clock. With

this sampling frequency, the A2D actually outputs 256 pixels per line. But in order to allow the RAM addresses one clock cycle for their addresses to settle before writing to them, only ever other pixel is stored.

Pixel Counters

Pixel counters are used as an interface between the VPU and the storage unit. The counters are made in VHDL and implemented in the CPLD. These counters determine where in RAM each pixel will be stored. A separate counter is used for the x value and the y value of each pixel. With an image resolution of 128x256 fifteen address bits are needed to address the storage. In the Shadowball addressing scheme, the top 8 bits correspond to the y value (line number) and the bottom 7 bits correspond to the x value of each pixel.

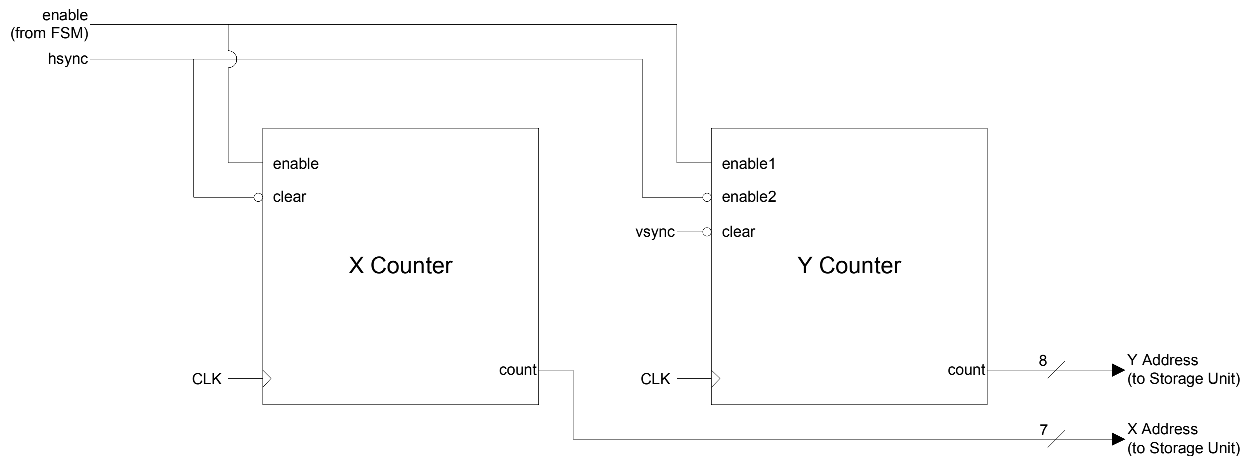


Figure 5: Pixel Counters

The x counter is clocked with the system clock and is enabled using an enable signal from the FSM. It is cleared by the hsync signal. The enable signal is active when the write enable to either RAM is active (more on this in storage section). While a frame is being inputted, the enable is active every other clock cycle. This is to allow the address bits to have time to settle before the RAM attempts to read or write. So every other pixel outputted by the A2D is stored. At the end of a line the counter clears, and stays cleared until the hysnc signal is over and the next horizontal line begins.

The y counter is also clocked by the system clock. But this is just because it is in the CPLD. It is enabled using both the enable from the FSM and the hsync signal. In essence, it could be clocked by the hsync signal so that the vertical counter increments at the end of each line. The y counter is cleared using the vsync signal. While vsync is low, the counter remains cleared. It starts counting again after vsync returns high and active video is being input.

Digital to Analog Converter

An Intersil 3338 high speed digital to analog converter is used to convert the digital values stored in the output buffer of the Storage Unit back into an analog form. Its analog output is sent directly to the RGB input of the video monitor. It is clocked with the 5MHz system clock and configured to continually convert on each clock cycle.

Storage Unit

The Storage unit is more complex than the video unit. The storage unit is designed to interface with both the video processing unit and the ball processing unit (BPU). It is composed of two 32 k RAM chips and logic to control all the addresses, data I/O, and control bits. Two RAM chips are used so that one chip can store data from the camera at the same time that the other outputs to the monitor. After each frame the two RAM chips swap roles. The BPU has control of the storage unit in the time between frames. During this time it can check on the values at pixel addresses and write the ball to the appropriate location. The BPU has bi-directional access to the storage unit. This is accomplished by having all the inputs and outputs of the RAM sent through muxes or switches. In this manner, both the VPU and BPU can treat the storage unit as if it were a single RAM attached directly to that unit. The control logic determines which RAM is being written to and which RAM is being read from. It also determines if the data, addresses and controls come from the VPU or the BPU. The control logic is implemented using an FSM with 6 states.

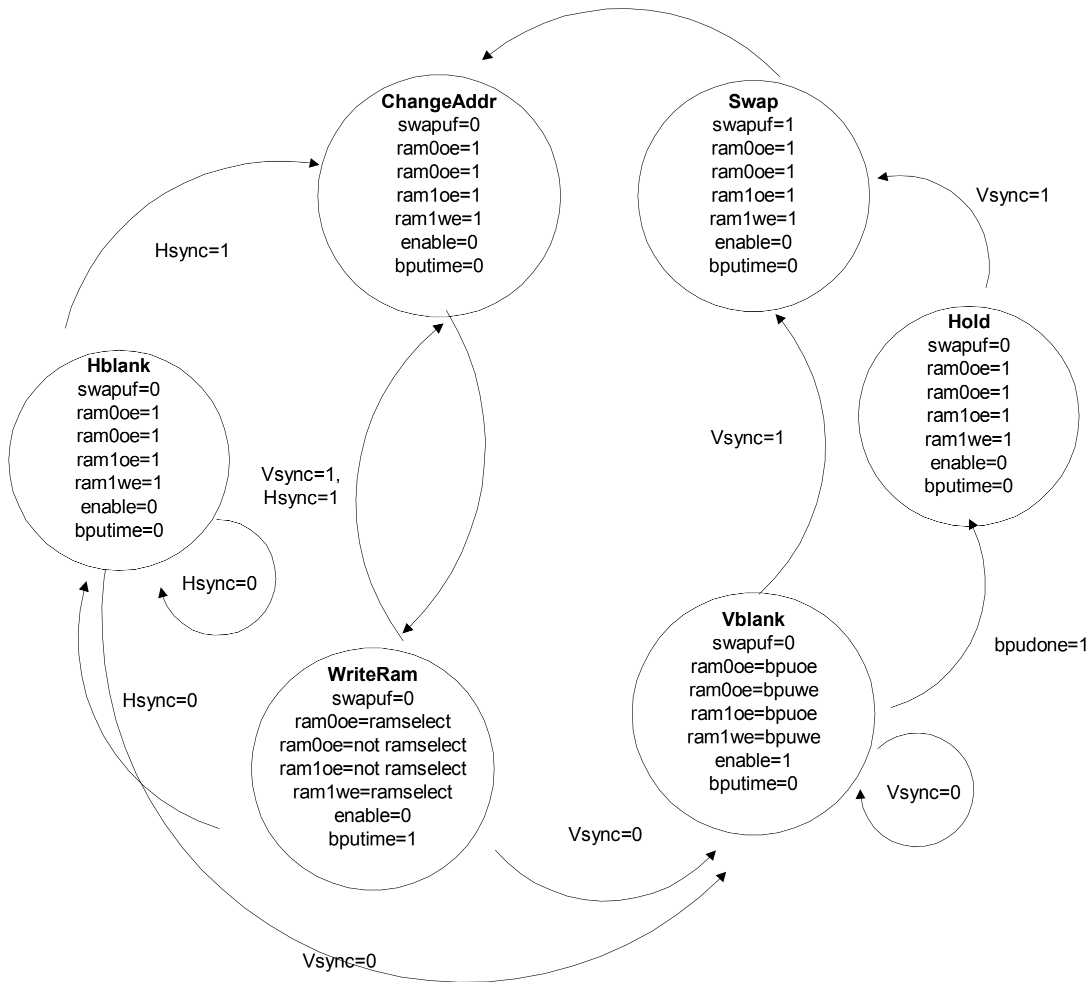


Figure 6: Storage Unit Control FSM

Control FSM

All of the control of the storage unit is done using a 6 state FSM (see figure x). When a frame is being input and output, the FSM goes between the ChangAddr, WriteRam, and Hblank states. The writing to and outputting from RAM happen in the WriteRam state. The Writeram

state also enables the address counters. This makes the address counters increment at the end of the WriteRam state. The ChangeAddr state is used to allow the address bits to settle before the RAM is read or written to. When an hsync signal comes in, the FSM goes to the Hblank state to disable all inputs and outputs as well as disabling the address counters. When a Vsync signal becomes active (active low), the FSM enters the Vblank state. During this time, the BPU has control of the storage unit. The control signals come from the BPU and the FSM outputs a BPUtime signal. This signal is sent to all the muxes and switches to ensure that data and addressing is controlled by the BPU lines. The BPU sends back a BPUdone handshaking signal when it has finished writing the new location of the ball to the RAM. At this time, the FSM enters the SwapBuf state which is used to Swap the functions of the two RAM chips. After this the FSM stays in a Hold state till the Vsync signal becomes inactive again, indicating the start of a new frame. If the BPU done signal does not come for some reason, the FSM will go straight from the Vblank state to the SwapBuf state to ensure that the video input and output will continue as normal.

RAM Addressing

Each RAM address corresponds to a pixel location in the frame. Each RAM address contains an 8 bit grayscale value for that pixel. With a resolution of 128x256, a 15 bit value addresses the 32 k RAM chip. As previously stated, the high order 8 bits are used as the y location and the lower 7 bits are used as the x location. When the VPU has control of the storage unit these address locations are calculated using the address counters in the VPU. When the BPU has control of the storage unit, the full 15 bit pixel address comes directly from the BPU. All

addresses to the RAM come from a mux that can select addresses generated by the X and Y counters or an address generated by the BPU. Both RAM chips use the same addresses since the video is output is output synchronized to the video being input.

Buffer Swapping

After each frame has been completed and after the ball has been written to the new frame, the two RAM chips swap functions. The new frame is outputted and the frame that was just outputted is rewritten. The buffer swapping logic takes care of swapping the RAMs and keeping track of which RAM is which. Swapping buffers is done using a T Flip Flop in the CPLD. The FSM outputs a SwapBuffer signal after the BPU has finished its ball processing. This causes the RamSelect signal to toggle. Whenever RamSelect is 1, the camera writes to Ram 0 and outputs from Ram 1. Whenever RamSelect is 0, the camera writes to Ram 1 and outputs from Ram 0. RamSelect is used as the selector for a multiplexor that takes in the data lines from both RAM chips and outputs from the appropriate RAM. The RamSelect is also used to control the write enable, output enable, and chip enable inputs to the RAM as indicated in the FSM (figure 6).

Data I/O

The data I/O is probably one of the most intricate parts of the entire Shadowball unit. Both the VPU and the BPU need to be able to input and output data from the correct RAM chip. The CPLD takes in an 8 bit data line from the A2D converter, an 8 bit data line from the BPU, and has an 8 bit data line to each RAM chip. Most of the intricate logic was performed using

VHDL in the CPLD. During the ChangeAddr and WriteRam states, the VPU is in control of the RAM. The RamSelect bit is used to determine which RAM chip the data from the camera is directed to. The CPLD outputs high impedance to the data lines of the other RAM chip.

When the BPU has control, it needs to be able to both read from and send data to the data lines. During the BPU state, when the BPU asserts write enable, the data line from the BPU is used as input and the data line to the appropriate RAM (as determined by RAMSelect) is used as output. When write enable is not enabled, the data line to the RAM is used as an input and the data line to the BPU is used as an output. The full VHDL used to produce this behavior is shown in Appendix D.

The data lines connected to the RAM chips are also connected to an external multiplexor. This 16:8 bit multiplexor is implemented using two 20v8 PAL chips. It uses the RAMSelect as its selector. This mux is used to choose the correct RAM to output from. The output is sent to the D2A which converts the 8 bit digital signal to an analog signal for the monitor.

Ball Processing Unit

Physics Algorithm

The physics algorithm is used to calculate the next position of the ball based on its current position, direction of motion, and surrounding objects. This algorithm determines when the ball should continue moving in its current direction as well as when it should bounce.

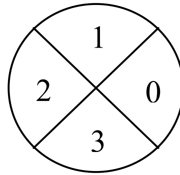


Figure 7: Ball Quadrant Encoding

The ball perimeter is divided into four quadrants illustrated above.

Any contact the ball has with its surrounding environment will be encoded as an intersection with one or more of these four quadrants. To detect ball contact, the points on the background image corresponding to the ball perimeter are checked. If any of the values at these points are greater than a set threshold value, that point is considered an intersection between the ball and its environment. Therefore, after all ball perimeter points are checked, a set of intersection points will be available. These intersection points are then mapped to the quadrant encoding described above. Namely, if any points in quadrant n (n=0,1,2,3) exhibit an intersection, quadrant n is flagged. This produces a set of four flags (one per quadrant) that characterizes the contact between the entire ball and its surrounding environment. For example, if represented as the four-bit number $Q_3Q_2Q_1Q_0$, 0110 would imply that quadrants 1 and 2 of the ball are in contact with the environment.

Once quadrant contact has been established, the algorithm can determine the next

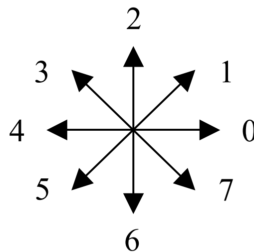


Figure 8: Ball Direction Encoding

direction and position of the ball based on its current direction. The ball direction is encoded as a number between 0 and 7 as in Figure 8.

A result of such encoding is the fact that there are a total of $2^4 = 16$ possible quadrant intersections and a total of 8 possible current directions. Therefore, there are only $16 \cdot 8 = 128$ possible different situations that can occur. Given one of these 128 possible situations (quadrant intersection along with currently direction), a new direction must be calculated. Since the number of possibilities is finite and quite small, this is accomplished using a lookup table. The lookup table used will provide a new direction for every possible situation.

More formally, if Q is the set of all possible quadrant intersections, D is the set of all possible directions, the lookup table provides the following mapping: $QXD \rightarrow D$.

The current ball direction and position is stored internally. In between each frame, the above algorithm is run to determine a new direction and position for the ball. These values are then stored as the current direction and position and the cycle repeats frame after frame.

Control MCU

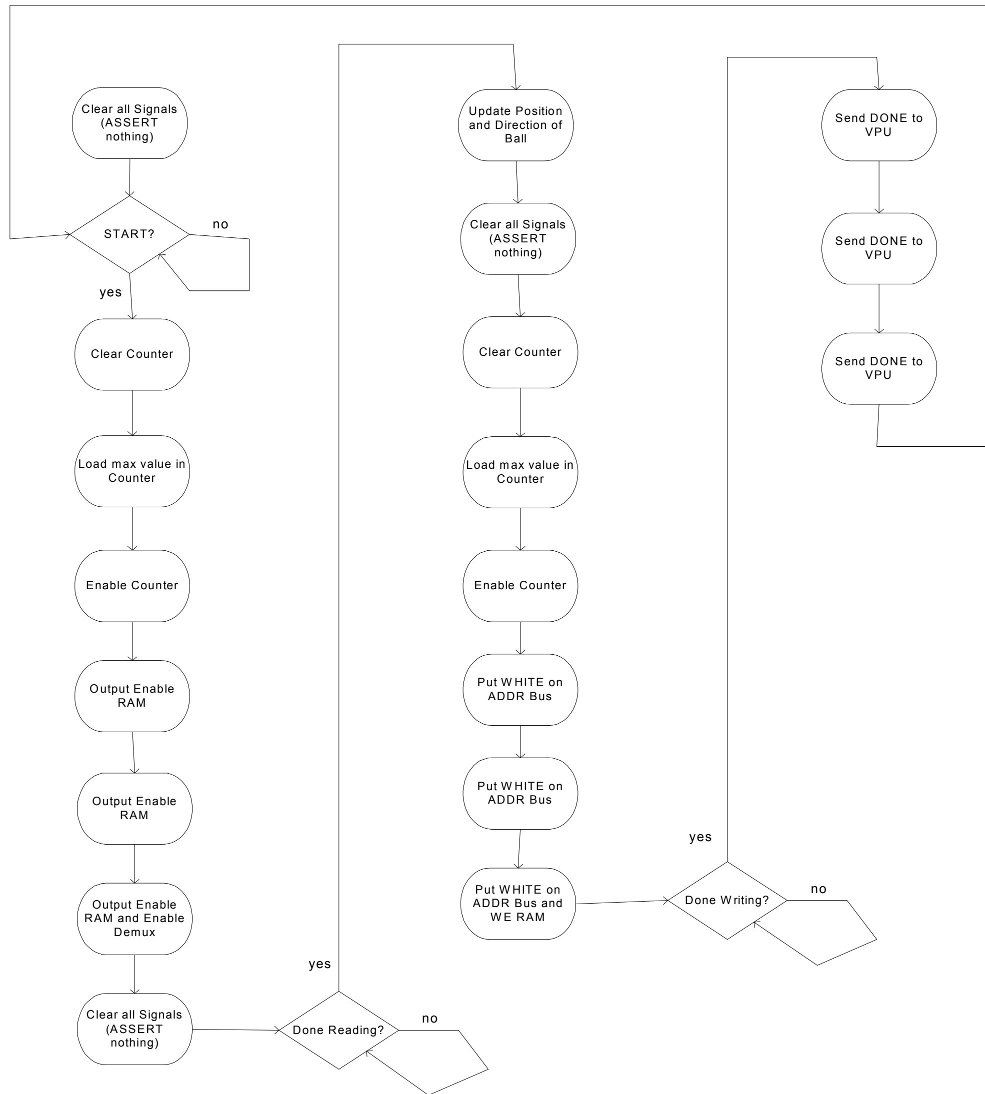


Figure 9: BPU Control MCU

Figure 9 shows the general flow of the microcontroller code for the ball-processing unit. The first step is to figure out if the ball is intersecting with any objects on the screen. This is done by sequentially running through each address of the ball PROM and passing the data

through the intersection detection unit. Once every pixel of the ball has been looked at, the values in the four registers, Q1-Q4 are valid. Thus the address to the lookup table is valid and an update is enabled. After this the counter is cleared and every single pixel is addressed again. But this time, instead of checking for intersection, a value of "11111111" is written to the RAM at that location. This creates the white outline of the ball that is seen on the screen.

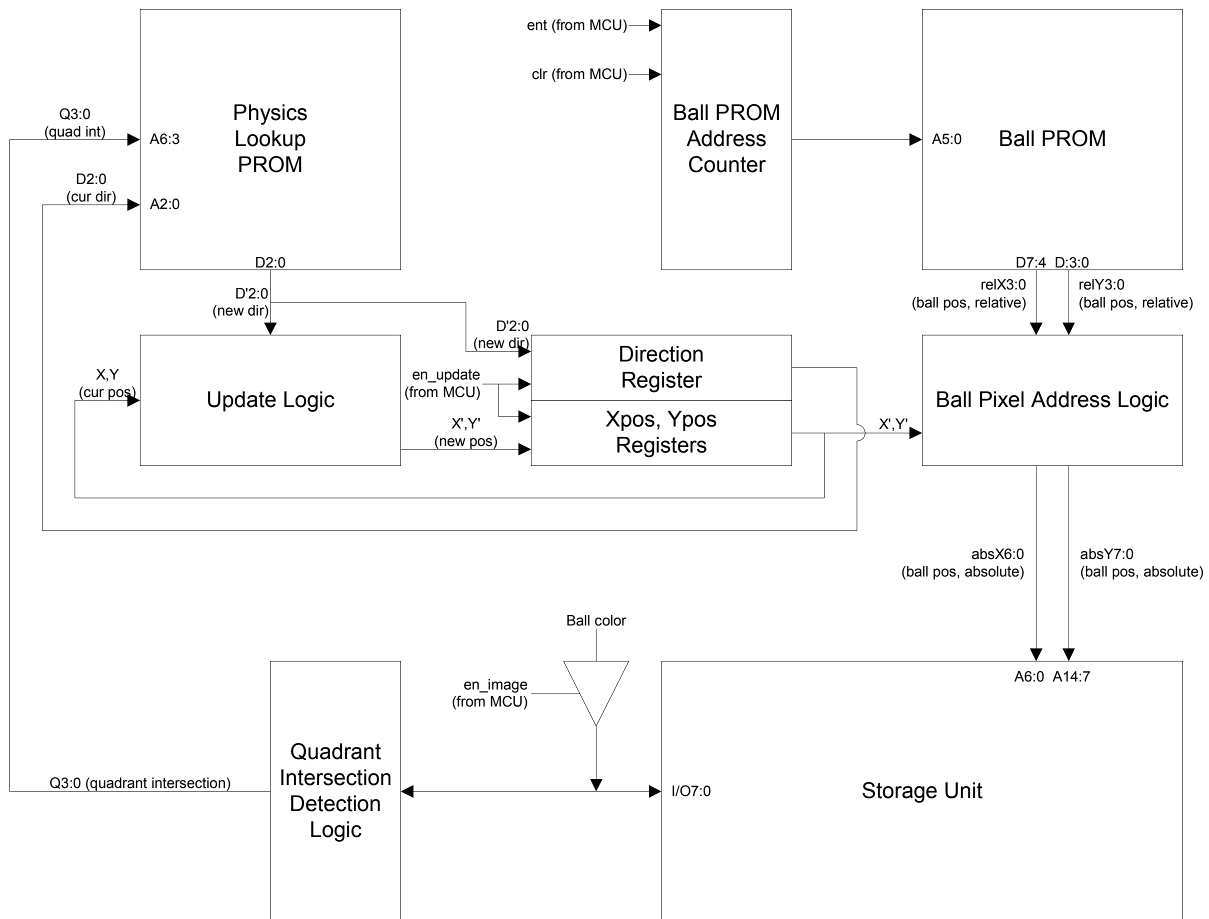


Figure 10: BPU Block Diagram

State Registers

The state of the ball is stored in three registers: direction, x position, and y position. The values of these registers are always interpreted as the *current* state of the ball. They are updated once in between each frame by the MCU with the *next* state of the ball. When the correct values for the next state are available at the inputs of the registers, the MCU asserts the en_update signal. This signal enables loading of the registers.

The direction register contains a 3-bit number representing the direction encoding described above. The x and y position registers are each 14 bits wide internally. However, of these 14 bits, the lower order 6 bits are interpreted as the fractional part of the number. The higher order 8 bits represent the whole number. When addressing the Storage Unit, only these high order bits are considered. The reason that this was done was to enable the ball to travel at speeds other than just integer values. This was in fact necessary to achieve a ball speed of 1 pixel per frame since any diagonal motion required incrementing the x and/or y registers by a values less than one (see the Update Logic Section below).

Ball PROM

The shape of the ball was burned onto a 28F256 PROM to allow for easy changes to ball shape and size. Each pixel of the ball is stored in the PROM as an 8-bit number representing the relative x and y positions of that pixel. The higher order 4 bits is the x position and the lower order 4 bits is the y position. All ball pixel positions are relative to the *actual* ball position which

is at the bottom left of the square binding the shape of the ball. A 4-pixel ball is given below as an example.

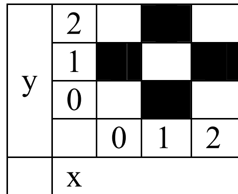


Figure 11: Ball PROM Pixel Location

The actual ball location is at (0,0) and all other pixels are specified in the PROM relative to that. The first location (address 0) of the PROM contains the number of pixels `n` the ball contains. The following `n` locations in the PROM contain the relative x and y positions of each pixel. By convention, all pixels are listed in the order corresponding to the quadrant that they belong in. Therefore, the above 4 pixel ball would have the representation given below in the ball PROM.

	Data	
Address	D7:4 (X3:0)	D3:0 (Y3:0)
0	4 (number of pixels, relevant lines to follow)	
1	2	1
2	1	2
3	0	1
4	1	0
5	…	…
…	…	…

Figure 12: Ball PROM Data Format

In the final implementation, the ball was much larger however the above example was used for early testing. The format however did not change since it allowed for a change in the size and shape of the ball without affecting the other components in the BPU.

The Ball PROM is addressed with a counter that is controlled by the MCU. When the first address of the PROM (address 0) is read, its value is stored in a register. The address bits going to the Ball PROM are compared with the value of this register as the address counter increments. When the address being outputted by the address counter equals that stored in the register, the signal done_rw is set to high and the MCU knows not to continue incrementing the Ball PROM address. At this point, each pixel of the ball has addressed (for either intersection detection or ball image writing). When the pixels of the ball have to addressed again, the address counter is cleared and process is repeated.

Since the values outputted by the Ball PROM are only relative to the actual ball position, they need to be processed by some logic (Ball Pixel Address Logic in Figure 10) before the absolute address of each pixel can be determined. Essentially, this logic just adds the relative positions given by the Ball PROM with the current position of the ball presented by the state registers. After doing so, the resultant x and y positions go into the corresponding address bits of the Storage Unit.

A more detailed logic diagram of this logic is shown below.

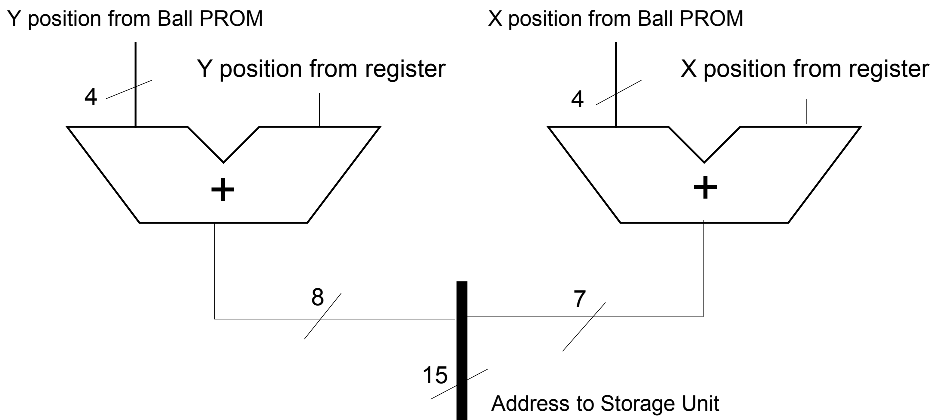


Figure 13: Ball Pixel Address Logic

Quadrant Intersection Detection Logic

There are four clearable, positive enabled registers in the system each of which represent intersection with one of the four quadrants of the ball. Their values at the end of looking at every pixel in the ball represent which quadrant intersected, if any. For example values like

Q1 = ‘0’

Q2 = ‘1’

Q3 = ‘0’

Q4 = ‘0’

represent that only quadrant 2 intersected with the ball.

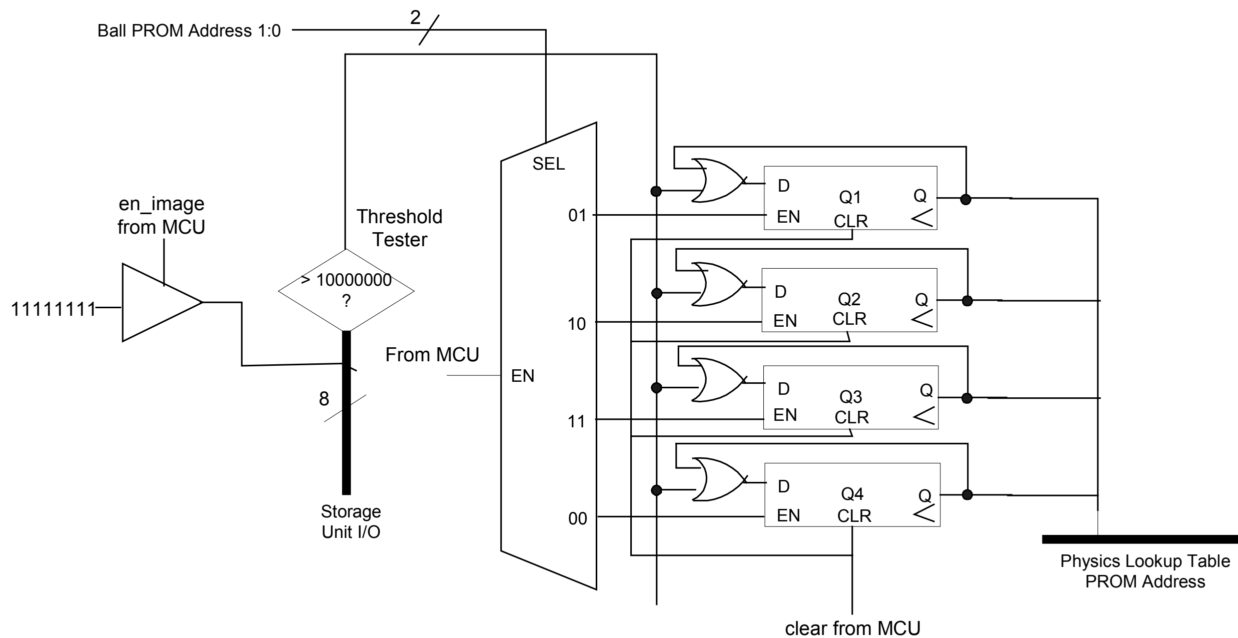


Figure 14: Quadrant Intersection Detection Logic

The values for the registers are stored sequentially as each pixel of the ball is read. First a pixel is addressed in the storage unit. The eight-bit data for this pixel is checked against the threshold, “10000000”. If the data is whiter (greater than) this value the output of the threshold decision unit is high, otherwise it is low.

Now a decision must be made as to where to store this data. Each pixel is in one of four quadrants. Thus the threshold value for one pixel must only be written to one of the four registers, Q1-Q4. This is done with the help of a demux and some addressing logic.

The 6 bits of address being used to address the Ball PROM are passed through some logic to reduce it to one of four quadrant values. The mapping is hard-coded into VHDL according to the layout of the ball in the PROM. The two-bit value coming out of this addressing logic is then used to control a demux which enables the proper register.

This process is probably better explained with the use of an example. Say the ball-pixel being addressed lies at address “001000” of the ball PROM and say this is in the third quadrant. “001000” comes out of the counter, goes through the addressing logic, and is mapped to a value of “11”. This “11” goes into the demux and sets the third line high, and all the other lines low. Thus on the next rising clock edge, only Q3 is enabled for a load. It will load its old value or-ed with the value coming out of the threshold unit. The or-ing is done because there are multiple pixels in each quadrant and an intersection is detected even if one of these pixels intersects.

Physics Lookup PROM

As described in the algorithm, the next direction of the ball is determined using a lookup table. This lookup table was implemented using 28F256 PROM. The current direction and quadrant intersection was encoded into the address of the PROM in the following way. The lower order 7 bits of the address were used; all other address bits were set to zero. Of these 7 bits, the lower order 3 bits represented the current direction and the higher order 4 bits represented the quadrant intersection. The values available at a particular address represented the next direction.

$Q_3Q_2Q_1Q_0D_2D_1D_0$ → $D'_2D'_1D'_0$

7 bit address 3 bit data

$Q_3Q_2Q_1Q_0$ Quadrant intersection: $Q_x = 1$ implies quadrant x is intersected, $Q_x = 0$ otherwise.

$D_2D_1D_0$ Current direction: 3 bit binary representation of the possible 8 directions.

$D'_2D'_1D'_0$ Next direction: 3 bit binary representation of the possible 8 directions.

An excerpt of the PROM data is reproduced below.

Address		Data
Quadrant Intersection	Current Direction	Next Direction
$Q_3Q_2Q_1Q_0$	$D_2D_1D_0$	$D'_2D'_1D'_0$
0000 (no intersection)	001 (1: 45° NE)	001 (1: 45° NE, no change)
0010 (quadrant 1)	010 (2: N)	110 (6: S, opposite direction)
…	…	…

Figure 15: Physics Lookup PROM Excerpt

The quadrant input is taken directly from the output of the four quadrant registers in the Quadrant Intersection Detection Logic and the direction is taken from the state registers. Thus, once these values are valid and available at the address of the PROM, the PROM's output will provide the next direction that the ball should travel in. This direction is passed to the Update Logic to determine the next position of the ball.

Update Logic

The calculation of the next ball position is based on the current position and the next direction. Since the velocity of the ball v_0 is always constant, it was hard-coded into the VHDL. Since the ball can move at an angle of 45° to the vertical or horizontal (directions 1,3,5,7), $v_0 /\sqrt{2}$ was also hard-coded into the VHDL. In the implementation, v_0 was set to 1. Therefore the ball moved at 1 pixel/frame.

Given the current position and the next direction, the PAL simply adds or subtracts either v_0 or $v_0 /\sqrt{2}$ to the current x and y positions to determine the new x and y positions. For example, if the current position is (x,y) and the next direction is 1 (45° NE), the new position calculated by the PAL would be $(x + v_0 /\sqrt{2}, y + v_0 /\sqrt{2})$.

Once the next direction and position are determined and available at the input of the direction and position registers, a signal en_update is sent from the MCU and these values are stored in the registers. At the next frame, these values will be interpreted as the “current” values and the ball will have moved.

Overall, once the each pixel of the ball is checked for intersection the four quadrant registers will contain valid data. Then the output of the Physics Lookup PROM will provide the next direction to the Update Logic. Shortly after that, the output of the Update Logic will be valid and contain the next the x and y position of the ball. Once this occurs, the MCU instructs the state registers to load new values (the next state).

Testing and Debugging

Ball Processing Unit

Testing of the BPU was initially a nontrivial task since the VPU and Storage Unit were being constructed simultaneously. Therefore, it was not possible to see the ball actually moving or bouncing on the screen. In fact, there was not even a background image that could be used for testing since such an image would require the Storage Unit to be functioning. Therefore, the only way the BPU was tested initially was by manually stepping through each line of microcode and simulating input signals. The program counter value was wired to HEX LEDs and all assertion signals were wired to normal LEDs. The clock was wired to a push button to allow for manual simulation. Though not ideal, such testing proved to be very helpful and gave very good results.

Once the VPU and Storage Unit were constructed, the entire system was connected and could be tested much more effectively. Once this was done many errors became apparent.

Moving Through Solid Objects

When first implemented, the Ball PROM contained a four-pixel ball (given above as an example of the Ball PROM format). This appeared a dot on the screen. The dot was observed to move correctly however, once it started to move in one direction, it never stopped moving in that direction. Even when it hit pixels that were clearly above the threshold value (thus should be interpreted as solid and should cause the ball to bounce off them), it continued to move straight through these pixel. The signals of BPU were observed using the logic analyzer and it was

discovered that the values being read from the Storage Unit were always high. As we later discovered, this was due to the fact that the Storage Unit was not correctly switching control to the BPU. This was a design flaw and changes to the Storage Unit were made. See below for more details.

Horizontal Jumping

The ball was observed to jump from place to place when moving horizontally. This phenomenon did not occur with vertical motion however. This suggested that it was a problem with the x addressing from the BPU to the Storage Unit. Further testing showed that this was caused by some logic error in the Storage Unit which resulted in the discarding of the lower order bits of the address from the BPU. Therefore, as the lower order bits of the x address were being changed, no change of the ball position was evident on the screen. Once sufficient lower order changes effected a higher order bit (that was not being discarded) a change on the screen would occur resulting in the jumping effect. Changing the necessary logic solved this problem.

Random Movement

After changing the ball size and shape to something significantly larger, a new error presented itself. When moving through a completely black background the ball should just travel straight. However, this new ball moved erratically in such a situation. After much testing and timing analysis the error was determined to be a problem with way the MCU was checking and writing the ball pixel values. Namely, the MCU checked a point of the ball for intersection and,

on the next clock cycle, wrote that point to the Storage Unit. Due to the way the Storage Unit was switching between write and read status, when writing to the RAM many other adjacent addresses were being written as well. Since other addresses are being written to with a value that is greater than the threshold, these pixels will be detected as intersections when checking future pixels of the ball. When the small "dot" ball was used this was not a problem since the there were only four pixels to checked. However, with the larger ball it became very noticeable since the likelihood of checking one of these miswrites was increased dramatically with the size of the ball. This problem, once discovered, was easily solve by rearranging some MCU microcode.

Downward Drift

Just as the entire system appeared to be working fully, a very small bug became evident. This error was very difficult to correct since it was, by its nature, very difficult to detect. The problem was that as the system ran, the ball would have a downward drift. Eventually the ball would be at the bottom of the screen always and it would be virtually impossible to cause it to move back up. Since the ball appeared to be bouncing and moving correctly, this was a very difficult problem to debug. Eventually however the problem was discovered to be a sign convention error made in the design. Namely, in the Ball PROM a certain sign convention was used for the y axis while, in the rest of the system, the opposite sign convention was used. Once this problem was discovered, its solution was trivial. However, discovering the problem, that was the hard part.

VPU and Storage Unit

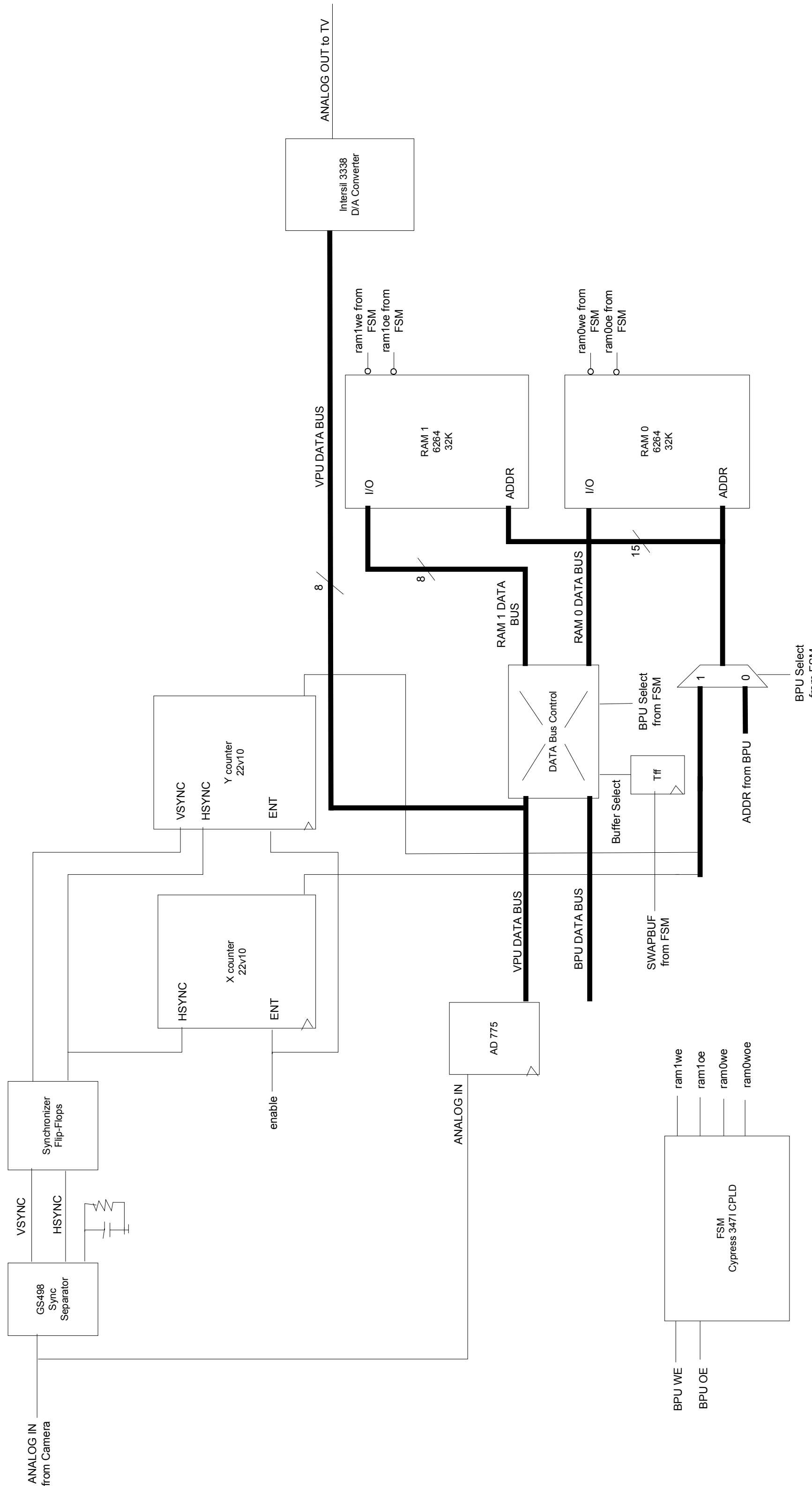
Due to the complexity of wiring involved, the VPU and Storage unit needed a significant amount of testing and debugging. The size of the busses and the need for everything to be muxed or switched meant that all of the logic could not be accomplished on the CPLD. There just weren't enough pins. So the address counters and muxes were done on external PALs and the output mux that fed the D2A was also done using external PALs. Small wiring mistakes caused a great deal of trouble and took time to debug. The two-way switching for the data lines also took a bit of debugging and playing with VHDL to figure out. We ended up needing to make nearly all of the buses into inouts and putting high impedance on the pins while inputting from them. In the end, the video still had a bit of jitter, but it was more than sufficient for our needs.

Conclusion

After much design, construction, testing and debugging, the end result was a system that met or exceeded every goal that was initially set. The ShadowBall system is a highly interactive environment where the users' physical motions affect the virtual motions of the ball within the system. The design implemented was a good balance between mathematical accuracy and human sensory perception. In other words, though it could have been more accurate, doing so would add unnecessary complexity. The goal of achieving a highly scalable and expandable design was also met. The ability for the ball shape and size to be changed with minimal effect on other components demonstrates the scalability and modularity of the system. The end result is a human

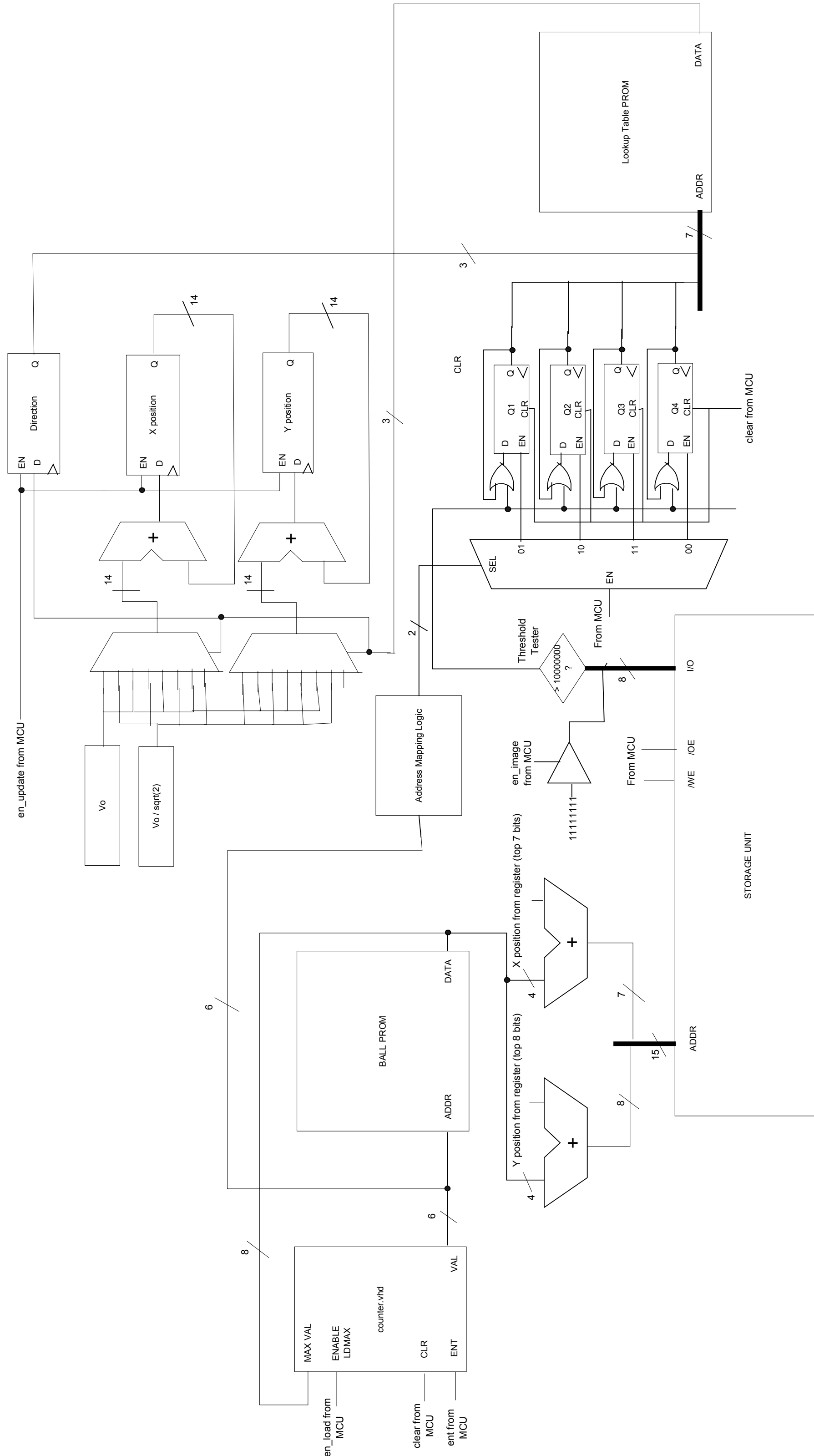
video interactive device that uses simple techniques and algorithms to achieve a highly realistic virtual world (fun to play with too).

Appendix A



Appendix A: Detailed Storage Unit Diagram

Appendix B



Appendix B: Detailed BPU Diagram

Appendix C

VPU – ram_io_switch.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity ramdatamux is

  port (
         selector   : in std_logic;
        ramselect: in std_logic;
        bpuwe: in std_logic;
        cameradata :in std_logic_vector(7 downto 0);
          bpudata:inout std_logic_vector(7 downto 0);
          ram1out: inout std_logic_vector(7 downto 0);
         ram0out : inout std_logic_vector(7 downto 0));
  end ramdatamux;


architecture mux of ramdatamux is
  signal vpu0out : std_logic_vector(7 downto 0);
  begin



data: process(ramselect, selector, bpuwe, bpudata, ram1out, cameradata, ram0out)
    begin

      if ramselect='0' then
        ram0out<="ZZZZZZZZ";

        if selector='1' and bpuwe='0' then
          bpudata<="ZZZZZZZZ";
          ram1out<=bpudata;
        elsif selector='1' and bpuwe='1' then
          ram1out<="ZZZZZZZZ";
          bpudata<=ram1out;
        else
          ram1out<=cameradata;
        end if;

      else
        ram1out<="ZZZZZZZZ";

        if selector='1' and bpuwe='0' then
          bpudata<="ZZZZZZZZ";
          ram0out<=bpudata;
        elsif selector='1' and bpuwe='1' then
          ram0out<="ZZZZZZZZ";
          bpudata<=ram0out;
        else
          ram0out<=cameradata;
        end if;

      end if;
    end process data;
end mux;
```

VPU – vpu.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity top is

  port (
    --  selector   : in std_logic;
    clk, vsync, hsync:         in std_logic;
    bpudone:            in std_logic;--bpu bits
    bpuwe:              in std_logic;
    bpuoe:              in std_logic;
    ramselect:          inout std_logic;
    cameradata :        in std_logic_vector(7 downto 0);
    bpudata:            inout std_logic_vector(7 downto 0);--put bpu data here
    ram0oe, ram1oe, ram0we, ram1we, enable : out std_logic;
    bputime:            inout std_logic;       --indicates bpu controlling memory
    ram0cs, ram1cs:     out std_logic;      --  add later
    pres: out std_logic_vector(3 downto 0);
    ram1out:            inout std_logic_vector(7 downto 0);
    ram0out :           inout std_logic_vector(7 downto 0));


    ATTRIBUTE pin_avoid of top :ENTITY is

--      " 1 2 11 21 22 32 42 43 44 53 63 64 74 83"& -- Vdd, Gnd, VPP

      " 12 19 73  "&      -- These pins are the interconnect bus
                          -- for CPLD 2, 3, and 4. They are Serial I/O
                          -- pins for CPLD 1.

      " 13         "&     -- This is I0-9. Can screw up the clock of C1. Be
                          -- careful when using this.

--        The CPLD has 4 clock pins that can also be used as input pins.
--        However, all of them are tied together.
--        The 4 clock pins are " 20 23 62 65 " .
--        Depending on your design, the programmer will assign of them
--        to be the clock input, and use the others as general-purpose inputs.
--        This can be quite frustrating.
--        We will thus disable 3 of the 4 and hope the compiler likes our
--        choice. If it doesn't, we will just have to pick another one.

--       Lets use clock 1 and disable clock 2,3, and 4.

         " 23 62 65 "&

--       If we need to use clock 2 : then use  " 20 62 65 "&
--       If we need to use clock 3 : then use  " 20 23 65 "&
--       If we need to use clock 4 : then use  " 20 23 62 "&



     " 14 35 41 51 72 "&  -- Used by Programmer. No external connection.


       "54  60 61 66 71  82";

  end top;


architecture fsm of top is
--------------------------------------------------------------------------------
 -- ramdatamux
  component ramdatamux

    port (
```

```
      selector:         in std_logic;
      ramselect:        in std_logic;
      bpuwe:            in std_logic;
      cameradata:       in std_logic_vector(7 downto 0);
      bpudata:          inout std_logic_vector(7 downto 0);--put bpu data here
      ram1out:          inout std_logic_vector(7 downto 0);
      ram0out :         inout std_logic_vector(7 downto 0));
  end component;

--   ------------------------------------------------------------------------------
--   --ram1datamux
--   component ram1datamux
--     port (
--       selector:         in std_logic;
--       ramselect:        in std_logic;
--       bpuwe:            in std_logic;
--       cameradata:       in std_logic_vector(7 downto 0);
--       bpudata:          inout std_logic_vector(7 downto 0);--put bpu data here
--       ram1out :         inout std_logic_vector(7 downto 0));
--   end component;
-- ---------------------------------------------------------------------------------



 -- type StateType is (changeaddr, writeram, hblank, vblank, swap, hold);
  constant changeaddr : std_logic_vector(2 downto 0) := "000";
  constant writeram : std_logic_vector(2 downto 0):="001";
   constant hblank : std_logic_vector(2 downto 0):="010";
  constant vblank : std_logic_vector(2 downto 0) := "011";
  constant hold : std_logic_vector(2 downto 0) := "100";
  constant swap : std_logic_vector(2 downto 0) := "101";


  signal swapbuf: std_logic;
   signal p_s, n_s : std_logic_vector(2 downto 0);
  signal intram0we, intram0oe, intram1we, intram1oe, muxsel : std_logic;
--signal vram1we, vram1oe, vram0we, vram0oe : std_logic;

begin

  ----------------------------------------------------------------------------
  --structural instantiation of components

  ram: ramdatamux port map(
    selector=>bputime, ramselect=>ramselect, bpuwe=>bpuwe,  cameradata=>cameradata,
bpudata=>bpudata, ram1out=>ram1out,  ram0out=>ram0out);

  ----------------------------------------------------------------------------
  statemach: process(ramselect, p_s, n_s, vsync, bpuoe, bpuwe, bpudone)  --does initial
                                                  --state matter?
  begin
    case p_s  is
      when changeaddr=>
        swapbuf<='0';
        intram0oe<='1';
        intram0we<='1';
        intram1oe<='1';
        intram1we<='1';
        muxsel<='0';
        enable<='0';
        bputime<='0';
        n_s<=writeram;

      when writeram =>
        swapbuf<='0';
        intram0oe<= ramselect;
        intram0we<=(not ramselect);
        intram1oe<=(not ramselect);
        intram1we<=ramselect;
        muxsel<='0';
        bputime<='0';
```

```
    enable<='1';                  --should counter enable happen during
                                    --write state or change state?
    if vsync = '0' then
      n_s <= vblank;
    elsif hsync='0' then
      n_s<=hblank;
      else

      n_s <= changeaddr;
    end if;

  when hblank=>
    swapbuf<='0';
    intram0oe<= '1';
    intram0we<='1';
    intram1oe<='1';
    intram1we<='1';
    muxsel<='0';
    bputime<='0';
    enable<='0';
     if vsync='0' then
       n_s<=vblank;
     elsif hsync='1' then
       n_s<=changeaddr;
     else
       n_s<=hblank;
     end if;


  when vblank =>
    swapbuf<='0';
    --doesn't matter, setting to inactive
    intram0oe<= '1';
    intram0we<= '1';
    intram1oe<= '1';
    intram1we<= '1';
    muxsel<='1';
    bputime<='1';
    enable<='0';

    if vsync='1'  then
      n_s<=swap;
    elsif bpudone='1' then
      n_s<=hold;
    else
      n_s<=vblank;
    end if;

  when hold=>
    swapbuf<='0';
    intram0oe<= '1';
    intram0we<='1';
    intram1oe<='1';
    intram1we<='1';
    muxsel<='0';
    enable<='0';
    bputime<='0';
    if vsync='1' then
      n_s<=swap;
    else
      n_s<=hold;
    end if;

  when swap=>
    swapbuf<='1';
    intram0oe<= '1'    ;
    intram0we<='1';
    intram1oe<='1';
    intram1we<='1';
    muxsel<='0';
    enable<='0';
```

```
        bputime<='0';
        n_s<=changeaddr;

      when others => null;
    end case;

  end process statemach;

  we_oe_mux: process(muxsel)
    begin
      if muxsel='0' then
        ram0we <= intram0we;
        ram0oe <= intram0oe;
        ram1we <= intram1we;
        ram1oe <= intram1oe;
      else
        ram0we <= bpuwe;
        ram0oe <= bpuoe;
        ram1we <= bpuwe;
        ram1oe <= bpuoe;
      end if;
    end process we_oe_mux;



  state_clocked: process(CLK)
  begin  -- process
    if rising_edge(clk) then
      p_s<=n_s;
    end if;
  end process state_clocked;

  flippy: process(swapbuf, clk)
  begin
    if rising_edge(clk) then
      if swapbuf='1' then
        ramselect<=not ramselect;
      else
        ramselect<=ramselect;
      end if;
    end if;
  end process flippy;

--  muxes: process(bputime)
--  begin
  ram1cs<= not ((bputime and ramselect) or (not bputime));
  ram0cs<= not ((bputime and (not ramselect)) or (not bputime));
--  end process muxes;
     pres(2 downto 0)<=p_s(2 downto 0);

end fsm;
```

VPU – xcounter.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
use work.bit_arith.all;

entity x_counter is

  port (
         clk, hsync, we, bpu   : in std_logic;
          bpuaddr :in std_logic_vector(6 downto 4);
         addr : out std_logic_vector(6 downto 4);
          caddrout: out std_logic_vector(3 downto 0));
  end x_counter;


    architecture x of x_counter is
          signal caddr: std_logic_vector(6 downto 0);

        begin
counter: process(clk)
    begin

        if rising_edge(clk) then
              if we='1' then


              if (hsync='0') then
                caddr<="0000000";

            elsif caddr="1111111" then
              caddr<=caddr;
                  else
                        caddr<=caddr+"0000001";
               end if;

                      end if;
    --put mux here

                end if;


    end process counter;


-- addr(0)<=(bpuaddr(0) and bpu) or (caddr(0) and (not bpu));
 --   addr(1)<=(bpuaddr(1) and bpu) or (caddr(1) and (not bpu));
  --  addr(2)<=(bpuaddr(2) and bpu) or (caddr(2) and (not bpu));
  --  addr(3)<=(bpuaddr(3) and bpu) or (caddr(3) and (not bpu));
   addr(4)<=(bpuaddr(4) and bpu) or (caddr(4) and (not bpu));
   addr(5)<=(bpuaddr(5) and bpu) or (caddr(5) and (not bpu));
   addr(6)<=(bpuaddr(6) and bpu) or (caddr(6) and (not bpu));
    caddrout(3 downto 0)<=caddr(3 downto 0);



end x;
```

VPU – ycounter.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
use work.bit_arith.all;

entity y_counter is

  port (
         clk, hsync, vsync, we  : in std_logic;
--          bpuaddr :in bit_vector(7 downto 0);
         addr : out bit_vector(7 downto 0));
  end y_counter;


    architecture y of y_counter is
          signal caddr: bit_vector(7 downto 0);
          signal flag : std_logic;
        begin
counter: process(clk)                 --note: not enabling on we now
    begin

        if rising_edge(clk) then
             if (hsync='0' and flag='0' and we='0') then
                   flag<='1';
                   if vsync='0' then
                        caddr<="00000000";
                   elsif caddr="11111111" then
                        caddr<=caddr;
                   else
                        caddr<=caddr+"00000001";
                   end if;
                   elsif hsync='1' and flag='1' then
                         flag<='0';
                   else

                   caddr<=caddr;
             end if;






                end if;



    end process counter;

      addr<=caddr;


end y;
```

Appendix D

Storage Unit – xmux.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity outmuxlow is

  port (
         bpuselect   : in std_logic;
          caddrx :in std_logic_vector(3 downto 0);
           bpuaddr :in std_logic_vector(3 downto 0);
         addrout : out std_logic_vector(3 downto 0));
  end outmuxlow;


architecture mux of outmuxlow is
  begin

huh: process(bpuselect)
    begin


    if bpuselect='1' then
  addrout<=bpuaddr;
else
  addrout<=caddrx;
end if;
    end process huh;

end mux;
```

Storage Unit – ymuxhigh.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity outmuxhigh is

  port (
         bpuselect   : in std_logic;
          bpuy :in std_logic_vector(7 downto 3);
           caddry :in std_logic_vector(7 downto 3);
         addrout : out std_logic_vector(7 downto 3));
  end outmuxhigh;


architecture mux of outmuxhigh is
  begin
huh: process(bpuselect)
    begin


if bpuselect='1' then
  addrout<=bpuy;
else
  addrout<=caddry;
end if;
    end process huh;


end mux;
```

Storage Unit – ymuxlow.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity outmuxlow is

  port (
         bpuselect   : in std_logic;
          caddry :in std_logic_vector(2 downto 0);
           bpuaddr :in std_logic_vector(2 downto 0);
         addrout : out std_logic_vector(2 downto 0));
  end outmuxlow;


architecture mux of outmuxlow is
  begin

huh: process(bpuselect)
    begin


    if bpuselect='1' then
  addrout<=bpuaddr;
else
  addrout<=caddry;
end if;
    end process huh;

end mux;
```

Storage Unit – outmuxhigh.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity outmuxhigh is

  port (
         selector   : in std_logic;
          ram0data :in std_logic_vector(7 downto 3);
           ram1data :in std_logic_vector(7 downto 3);
         dataout : out std_logic_vector(7 downto 3));
  end outmuxhigh;


architecture mux of outmuxhigh is
  begin
huh: process(selector)
    begin


if selector='1' then
  dataout<=ram1data;
else
  dataout<=ram0data;
end if;
    end process huh;

end mux;
```

Storage Unit – outmuxlow.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;


entity outmuxlow is

  port (
         selector   : in std_logic;
          ram0data :in std_logic_vector(2 downto 0);
           ram1data :in std_logic_vector(2 downto 0);
         dataout : out std_logic_vector(2 downto 0));
  end outmuxlow;


architecture mux of outmuxlow is
  begin

huh: process(selector)
    begin


    if selector='1' then
  dataout<=ram1data;
else
  dataout<=ram0data;
end if;
    end process huh;

end mux;
```

Appendix E

BPU – address.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity address_RAM is

  port (
    y_pos, x_pos : in std_logic_vector(7 downto 0);
    update : in std_logic_vector(7 downto 0);
    address: out std_logic_vector(14 downto 0));
end address_RAM;

architecture behavioral of address_RAM is

  signal int_yaddr, int_xaddr: std_logic_vector(7 downto 0);
begin  -- behavioral

  int_yaddr <= y_pos + ("0000" & update(3 downto 0));
  int_xaddr <= x_pos + ("0000" & update(7 downto 4));

  address <= int_yaddr(7 downto 0) & int_xaddr(6 downto 0);

end behavioral;
```

BPU – counter.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity counter is

  port (
    clk      : in  std_logic;
    load     : in  std_logic_vector(5 downto 0);
    en_ldmax : in  std_logic;
    clr      : in  std_logic;
    ent      : in  std_logic;
    done     : out std_logic;
    value      : out std_logic_vector(5 downto 0));

end counter;

architecture behavioral of counter is

  signal maximum : std_logic_vector(5 downto 0);
  signal int_val : std_logic_vector(5 downto 0);
begin  -- behavioral

  -- NOTE: The clear takes the counter value back to "000000", but this is NOT
  -- the value for the first x,y data block in the PROM. Address "000000"
  -- instead holds the MAX VAL that should be loaded into the the maximum of
  -- this counter.  Then the counter should be incremented to go to address "
  -- 000001" where the first pixel of x,y data is stored.
  -- Thus basically for every update of the ball the maximum value is reloaded.
  -- This is both for purposes of simplicity and future expalnsion. This way if
  -- we had a switch to change balls, we could change the ball while its moving.

  clocked: process (clk)
  begin  -- process clocked
    if rising_edge(clk) then

      if clr = '1' then
        int_val <= "000000";
        maximum <= maximum;
      elsif clr = '0' and en_ldmax = '1' then
        maximum <= load;
        int_val <= int_val;
      elsif clr = '0' and en_ldmax = '0' and ent = '1' then
        int_val <= int_val + 1;
        maximum <= maximum;
      else
        int_val <= int_val;
        maximum <= maximum;
      end if;
    end if;
  end process clocked;

  done: process (int_val, maximum)
  begin  -- process done
    if int_val = maximum then
      done <= '1';
    else
      done <= '0';
    end if;
  end process done;

  value <= int_val;


end behavioral;
```

BPU – threshold.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity threshold is

  port (
    clk                         : in  std_logic;
    clr                         : in  std_logic;
    addr                        : in  std_logic_vector(4 downto 0);
    data                        : in  std_logic_vector(7 downto 0);
    en_demux                    : in  std_logic;
    q_out                       : out std_logic_vector(3 downto 0));

end threshold;

architecture behavioral  of threshold is

  constant THRESHOLD : std_logic_vector(7 downto 0) := "10000000";  --threshold
                                                                    --for dark
  signal demux_ctl : std_logic_vector(1 downto 0);
--object
  signal en_q: std_logic_vector(3 downto 0);
  signal d : std_logic;
  signal q: std_logic_vector(3 downto 0);
begin  -- behavioral

--***********************************************************************
  -- Quadrant Registers: There are four registers in the system, each
  -- corresponding to the intersection of one of the four points that make up
  -- the ball.
  -- Each register has an enable that is hooked up to the outputs of the demux
  -- below.  The registers only loads on the rising clock edge if the enable is
  -- active.
  --***********************************************************************

clocked: process (clk)
 begin  -- process clocked

   if clr = '1' then then
     q <= "0000";

   elsif rising_edge(clk) then

     if en_q(0) = '1' then
         q(0) <= d or q(0);
       else
         q(0) <= q(0);
       end if;

     if en_q(1) = '1' then
       q(1) <= d or q(1);
     else
       q(1) <= q(1);
     end if;

     if en_q(2) = '1' then
       q(2) <= d or q(2);
     else
       q(2) <= q(2);
     end if;

     if en_q(3) = '1' then
       q(3) <= d or q(3);
     else
       q(3) <= q(3);
     end if;
```

```
    end if;

  end process clocked;

  --*****************************************************************************
  -- Demux: This demux controls the enables of the four registers above. The demux_
  -- ctl signal determines which line of the demux is to be pulled high.  The
  -- demux in only active if the en_demux signal from the mcu is active.  If the
  -- enable is not active then all the output line from the demux will be low (i.
  -- e.none of the registers will be enabled to load).
  --*****************************************************************************

  demux: process (demux_ctl, en_demux)
  begin  -- process demux

    if en_demux = '1' then

      case demux_ctl is

        when "01" =>
          en_q <= "0001";

        when "10" =>
          en_q <= "0010";

        when "11" =>
          en_q <= "0100";

        when "00" =>
          en_q <= "1000";

        when others => null;
      end case;

    else
      en_q <= "0000";

    end if;
  end process demux;

  --*****************************************************************************
  -- Threshold Unit:  Simple threshold unit that sets the value for d, depending
  -- on the data being higher or lower than the threshold.  I am assuming that
  -- darker means a higher value, so the threshold is checked with a > symbol.
  --*****************************************************************************
  thresh: process (data)
  begin  -- process thresh
    if data > THRESHOLD then
      d <= '1';
    else
      d <= '0';
    end if;
  end process thresh;

  addr: process (addr)
  begin  -- process addr
    if addr < "00110" then
      demux_ctl <= "01";
    elsif addr < "01011" then
      demux_ctl <= "10";
    elsif addr < "10000" then
      demux_ctl <= "11";
    else
      demux_ctl <= "00";
    end if;
  end process addr;

  q_out <= q;

end behavioral ;
```

BPU – update.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity update_pos is

  port (
    clk, update_in, reset: in std_logic;
    dir_in : in std_logic_vector (2 downto 0);
    xpos_out, ypos_out : out std_logic_vector (7 downto 0);
    dir_out : out std_logic_vector (2 downto 0)
    );
end update_pos;

architecture arch of update_pos is

  signal xpos, ypos : std_logic_vector (13 downto 0);
  signal dir : std_logic_vector (2 downto 0);
  constant v : std_logic_vector (13 downto 0) := "00000001000000"; -- 1
  constant vr2 : std_logic_vector (13 downto 0) := "00000000101101"; --1/root2

begin
  main: process (clk, reset)
  begin
    if rising_edge (clk) then
      if reset = '1' then
        xpos <= "01000000000000";
        ypos <= "10000000000000";
        dir <= "010";
      elsif update_in = '1' and reset = '0' then
        dir <= dir_in;
        if dir_in = "000" then
          xpos <= xpos + v;
          ypos <= ypos;
        elsif dir_in = "001" then
          xpos <= xpos + vr2;
          ypos <= ypos + vr2;
        elsif dir_in = "010" then
          xpos <= xpos;
          ypos <= ypos + v;
        elsif dir_in = "011" then
          xpos <= xpos - vr2;
          ypos <= ypos + vr2;
        elsif dir_in = "100" then
          xpos <= xpos - v;
          ypos <= ypos;
        elsif dir_in = "101" then
          xpos <= xpos - vr2;
          ypos <= ypos - vr2;
        elsif dir_in = "110" then
          xpos <= xpos;
          ypos <= ypos - v;
        elsif dir_in = "111" then
          xpos <= xpos + vr2;
          ypos <= ypos - vr2;
        end if;

      else
        xpos <= xpos;
        ypos <= ypos;
        dir <= dir;
      end if;
    end if;
  end process main;

  xpos_out <= xpos (13 downto 6);
  ypos_out <= ypos (13 downto 6);
  dir_out <= dir;
```

```
end arch;
```

BPU – bpu_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity bpu is

  port (
    clk         : in    std_logic;
    reset       : in    std_logic;
    ld_max      : in    std_logic;      -- MCU assertion
    clr         : in    std_logic;      -- MCU assertion
    ent         : in    std_logic;      -- MCU assertion
    en_image    : in    std_logic;      -- MCU assertion
    en_demux    : in    std_logic;      -- MCU assertion
    en_update   : in    std_logic;      -- MCU assertion
    done        : out   std_logic;  -- MCU status signal
    ball_addr   : out   std_logic_vector(5 downto 0);  -- to ball PROM
    ball_data   : in    std_logic_vector(7 downto 0);  -- from ball PROM
    RAM_addr    : out   std_logic_vector(14 downto 0);  -- to Storage Unit
    RAM_io      : inout std_logic_vector(7 downto 0);  -- to/from Storage Unit
    lookup_addr : out   std_logic_vector(6 downto 0);  --to lookup table PROM
    lookup_data : in    std_logic_vector(2 downto 0));  -- from lookup table PROM

end bpu;

architecture structural of bpu is

 component counter
   port (
     clk      : in  std_logic;
     load     : in  std_logic_vector(5 downto 0);
     en_ldmax : in  std_logic;
     clr      : in  std_logic;
     ent      : in  std_logic;
     done     : out std_logic;
     value      : out std_logic_vector(5 downto 0));
 end component;

 component address_RAM
   port (
     y_pos, x_pos : in std_logic_vector(7 downto 0);
     update : in std_logic_vector(7 downto 0);
     address: out std_logic_vector(14 downto 0));
 end component;

 component threshold
   port (
     clk                        : in  std_logic;
     clr                        : in  std_logic;
     addr                       : in  std_logic_vector(4 downto 0);
     data                       : in  std_logic_vector(7 downto 0);
     en_demux                   : in  std_logic;
     q_out                      : out std_logic_vector(3 downto 0));
 end component;

 component update_pos
   port (
     clk, update_in, reset: in std_logic;
     dir_in : in std_logic_vector (2 downto 0);
     xpos_out, ypos_out : out std_logic_vector (7 downto 0);
     dir_out : out std_logic_vector (2 downto 0)
     );
 end component;

 signal int_ball_addr: std_logic_vector(5 downto 0);
 signal int_xpos, int_ypos: std_logic_vector(7 downto 0);
 signal int_q : std_logic_vector(3 downto 0);
 signal int_dir: std_logic_vector(2 downto 0);
```

```
begin  -- structural

  counter_unit: counter port map (clk => clk, load => ball_data(5 downto 0), en_ldmax => ld_max,
clr => clr, ent => ent, done => done, value => int_ball_addr);

  address_unit: address_RAM port map (y_pos => int_ypos, x_pos => int_xpos, update => ball_data,
address => RAM_addr);

  threshhold_unit: threshold port map( clk => clk, clr => clr, addr => int_ball_addr(4 downto 0),
data => RAM_io, en_demux => en_demux, q_out => lookup_addr(6 downto 3));

  update_unit: update_pos port map(clk => clk, update_in => en_update, reset => reset, dir_in =>
lookup_data, xpos_out =>int_xpos, ypos_out => int_ypos, dir_out => lookup_addr(2 downto 0));

  ball_addr <= int_ball_addr;

  -- TriState on Data Bus: When enabled, drives all data lines to +5V (high)

  tristate: process
  begin
    if en_image = '1' then
      RAM_io <= "11111111";
    else
      RAM_io <= "ZZZZZZZZ";
    end if;
  end process tristate;

end structural;
```

Appendix F

BPU Microcode – bpu_mcu.sp

```
/****************************************************************/
/* Instruction Word Organization:                               */
/*   conditional branches            0cccxxxx aaaaaaaa          */
/*   unconditional branches          0111xxxx aaaaaaaa          */
/*   assertion statements            1sssssss ssssssss          */
/*    where c = status selection                                */
/*          a = alternative address, i.e. jump address          */
/*          s = assertion signals                               */
/****************************************************************/

op <15:0>;                      /* Indicates the available bits              */
address op <7:0>;               /* Indicates bit locations for addresses     */
value op <7:0>;

/*
 * There is nothing magic about upper case.
 * You may change things to lower case as you wish.
 * Remember, the assembler maps all characters to lower case anyway!
 */

/*
 * Instruction set for your MCU
 */

CJMP   op<15>=%b0;    /* Conditional JuMP */
JMP    op<15:12>=%b0111;      /* unconditional JuMP */
ASSERT op<15>=%b1;    /* unconditional ASSERT */

/* These are defined so that you may use them to make your code more
 * readable.  Their use is not required. */

IF      nop;
THEN    nop;
TRUE    op<14:12>=%b111;        /* This causes the 151 to output true */
RESET   op<15:0>=%b0111000000000000;

assert_low op<7:6>;

/* Assertions */

ld_max          op<0>=1;
clr             op<1>=1;
ent             op<2>=1;
en_image               op<3>=1;
en_demux               op<4>=1;
en_update              op<5>=1;
we              op<6>=0;
oe              op<7>=0;
done            op<8>=1;

/*
 * Status signals:  Switches and frequency divider output OSC
 * Make sure that all status signals that change during mcu operation
 * are synchronized to the system /CLK
 */

start           op<14:12>=0;
done_rw         op<14:12>=1;

/*
From lab3
status_a2d      op<14:12>=0;
samp_clk               op<14:12>=1;
pass_shift      op<14:12>=2;
pass_orig              op<14:12>=3;*/
```

BPU Microcode – bpu_mcu.as

```
/*************************************************************/
/* Instruction Word Organization:                            */
/*   conditional branches             0cccxxxx aaaaaaaa      */
/*   unconditional branches           0111xxxx aaaaaaaa      */
/*   assertion statements             1sssssss ssssssss      */
/*    where c = status selection                             */
/*          a = alternative address, i.e. jump address       */
/*          s = assertion signals                            */
/*************************************************************/

op <15:0>;                      /* Indicates the available bits              */
address op <7:0>;               /* Indicates bit locations for addresses     */
value op <7:0>;

/*
 * There is nothing magic about upper case.
 * You may change things to lower case as you wish.
 * Remember, the assembler maps all characters to lower case anyway!
 */

/*
 * Instruction set for your MCU
 */

CJMP   op<15>=%b0;    /* Conditional JuMP */
JMP    op<15:12>=%b0111;      /* unconditional JuMP */
ASSERT op<15>=%b1;    /* unconditional ASSERT */

/* These are defined so that you may use them to make your code more
 * readable.  Their use is not required. */

IF      nop;
THEN    nop;
TRUE    op<14:12>=%b111;        /* This causes the 151 to output true */
RESET   op<15:0>=%b0111000000000000;

assert_low op<7:6>;

/* Assertions */

ld_max         op<0>=1;
clr            op<1>=1;
ent            op<2>=1;
en_image               op<3>=1;
en_demux               op<4>=1;
en_update              op<5>=1;
we             op<6>=0;
oe             op<7>=0;
done           op<8>=1;

/*
 * Status signals:  Switches and frequency divider output OSC
 * Make sure that all status signals that change during mcu operation
 * are synchronized to the system /CLK
 */

start          op<14:12>=0;
done_rw        op<14:12>=1;

/*
From lab3
status_a2d     op<14:12>=0;
samp_clk               op<14:12>=1;
pass_shift     op<14:12>=2;
pass_orig              op<14:12>=3;*/
```