“Squash Yourself”
6.111 Final Project Report

Will Fotsch, Azadeh Moini, Sumit Khatod
December 13, 2006

TA: Cassie Huang

Abstract

“Squash Yourself” is a two-player squash game. The players have paddles whose positions are determined by a filtered video input. The screen displays the squash court, paddles, and the ball as it would appear to the player. Additionally, the screen displays the ball’s heat, the players’ scores and the number of hits in the current rally. The ball increases in size as it approaches the player, and decreases in size as it moves back towards the virtual walls. A hit is determined by a swing of the player’s paddle if the ball is a certain size (is within the specified hitting range). If the player double-hits or misses the ball with a swing, the player loses the point or the serve. A player can only earn a point on his/her serve. The angle and power of a hit are determined by the speed and direction of the players’ paddles. Additionally, the harder the ball is hit, the hotter it gets, and the faster it moves. When one player accumulates fifteen points, the other player is “TERMAN-ATED,” and the game ends.

Contents

List of Figures

List of Tables

1. Introduction

For our final project, we created a two-player video input squash game. The project was partially motivated by Nintendo Wii's innovative approach to making gaming more realistic. We were also motivated by the personal desire to create an interactive game. We wanted to create a game that we could appreciate and enjoy playing ourselves. Since sports are something of interest to us and we find using a real paddle more interactive than pressing buttons, we thought this would be fun.

We found it most practical to divide the game into three basic sections: user inputs, game logic, and outputs.

To input paddle locations, we use a camera. The camera's YCrCb data is converted to RGB data and stored as 18-bit data in the ZBT. The data is then processed in a video input processing unit. This unit selects a certain pixel on the paddle, sets threshold noise levels for the pixel, and filters through the image to get a clear image of the paddle. By averaging these filtered pixel locations, the processing unit determines the center of mass of each paddle and displays them on the screen. The processing unit also looks at the last few frames of each swing to determine paddle direction and power (acceleration). These are output to the game logic modules and determine the speed and direction of the ball on the screen.

The game logic modules control the ball movement, detect collisions, determine whether a point is over, and tracks player scores. In order to simulate three-dimensional movement, the ball increases in size and moves down the screen slightly as it moves towards the player, and decreases in size and moves up the screen as it moves away from the player towards the back wall. The walls and ground are sectioned. In the sections closest to the edges (closest to the player), the ball has to be bigger to bounce back. The closer the ball is to the back wall, the ball will only bounce back only if it is smaller. Collisions are only detected if the ball is moving towards the user and the radius is within a certain range. Hit priority is given to the player that should be hitting.

If the radius increases beyond the maximum threshold without being hit, the point is over. Additionally, if the ball is double-hit by a player, the point is over. A player can only earn a point on his serve. If a player loses the point on their serve, they lose the serve to the other player. The point tracker adds up points, and when one player score reaches 15, the game is over. A rally counter keeps track of the number of hits per point.

The video output is a layered display of the court, ball, paddles, and scoreboard. The display was created blob by blob, and individual modules were created for each shape and reused to make various output components. The scoreboard consists of an increasing rectangular bar for ball heat, and a series of seven-segment displays for the player scores and rally. The paddles are transparent so the player can always see the ball. Additionally, an end-game screen appears when the game is over. All modules were pipelined to minimize the number of calculations per clock cycle.

Audio output was similar to lab 4. Eight BRAMs were created, one per sound. The audio is programmed with switches on the FPGA. The audio output allows the user to program the FPGA to announce hits, power hits, game reset, indicate serves and game winners, and allow players to taunt one another.

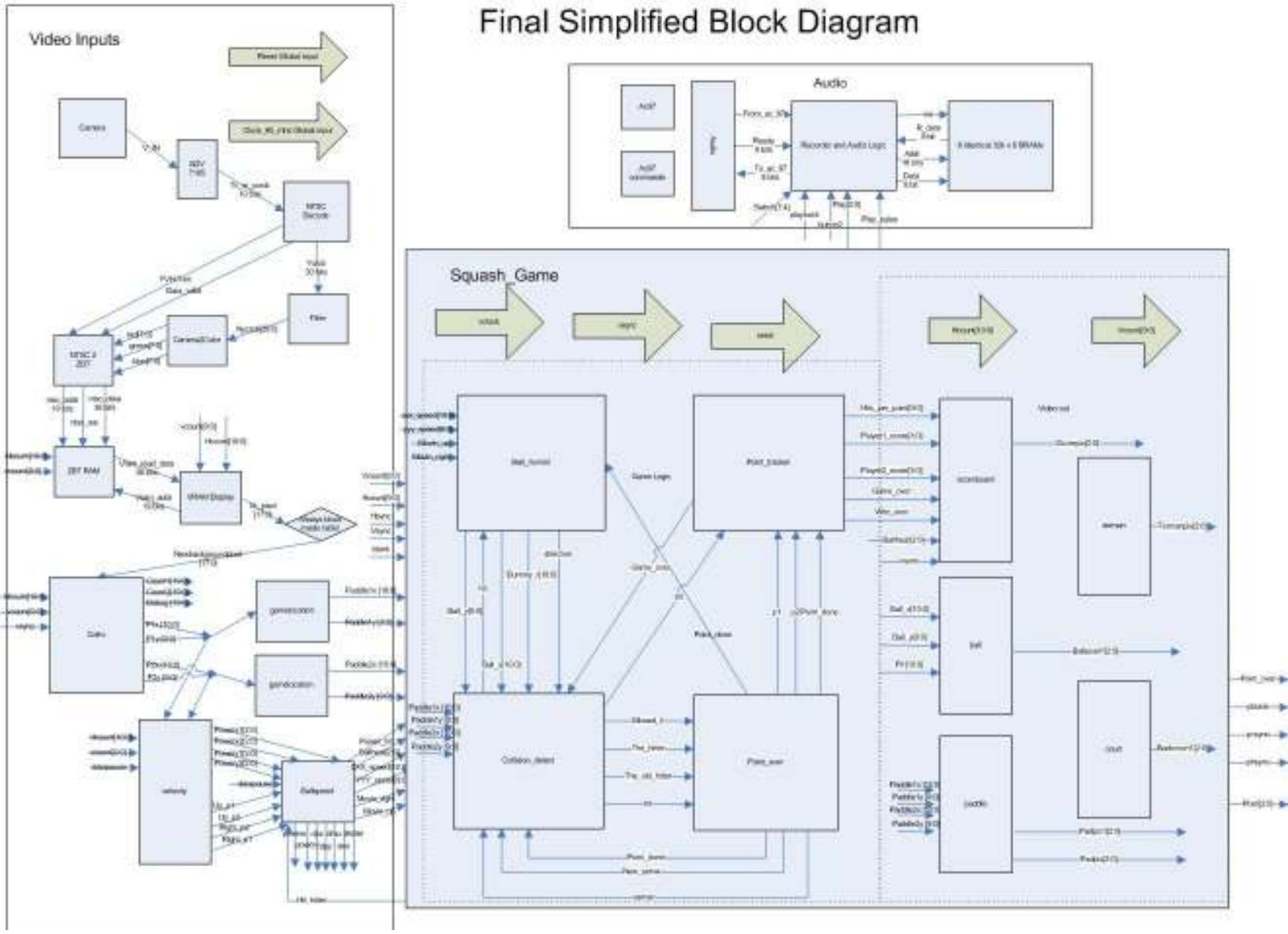

Figure 1.1 – Block Diagram

2. User Inputs
Sumit Khatod

The inputs to the squash game are handled via a video camera that tracks the position of the player's paddles by keying in on a specific color range. The players can then move the paddles controlling location of the paddle and the power of a swing. In order to create

this functionality, the video input portion of the lab is broken into three segments, the camera input, the video processing, and ballspeed.

2.1 Camera Input

The camera input segment is responsible for creating a pixel stream for the video processor to run calculations on. It consists of five main modules, a ZBT memory, a camera, the adv7185 chip on the labkit, and a small block in the labkit file. Figure 2.1 outlines the data path of this segment.

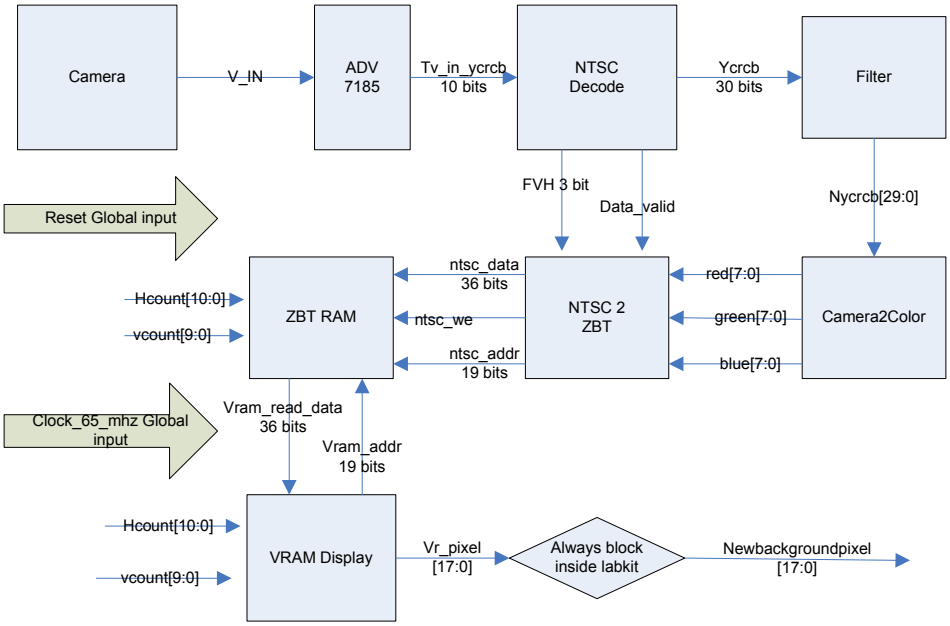

Figure 2.1: Camera input block diagram

The data from the camera is decoded by the adv7185 chip using the YCrCb color scheme. In this color scheme, Y represents luminance and Cr and Cb represent red and blue chrominance respectively. Using the staff provided ntsc_decode module, the data from the adv7185 is converted into a 30 bit YCrCb pixel stream that holds the 10 bits of each Y, Cr, and Cb. This module is needed because the data from the adv7185 comes in 10 bit packets rather than a constant stream changing only at each pixel.

After being decoded the YCrCb pixel stream goes through the filter module. This module looks for two color ranges, bright orange and light blue. If the current pixel is in the specified orange color range representing player one's paddle, it is changed to a white pixel. If the current pixel is in the specified blue color range representing player two's paddle, it is changed to a blue pixel. Otherwise, if the pixel is in neither color range, it is set to black. The pixel change allows for each color to be easily identifiable later by the video processing segment. The various thresholds for the color pixels were tested and determined using a calibration module that ran through possible filters. Furthermore, it is important to note that

color filtering occurs in the YCrCb color scheme rather than in the RGB. The motivation for this decision is that the YCrCb color scheme is more immune to intensity changes. Thus, by ignoring Y and focusing on Cr and Cb, the filter is significantly less sensitive to shadows and light changes.

The filtered data then travels to the color conversion module. This module was provided by xillinx and converts YCrCb signals into RGB signals using the formulas:

$$R' = 1.164(Y'– 64)+ 1.596(Cr – 512)$$
$$G' = 1.164(Y' – 64)–(0.813)(Cr – 512)– 0.392(Cb – 512)$$
$$B' = 1.164(Y' – 64)+ 2.017(Cb – 512)$$

The ten bit YCrCb is broken into 8 bits of red, 8 bits of blue, and 8 bits of green.

The new RGB pixel data is then placed into the ZBT by the ntsc2zbt module. The staff provided an initial version of this module to store 4 pixels per address in 6 bit of Y in ycrcb color space. However, the version used is heavily modified to hold 2 pixels per address each with 6 bits of red, green, and blue. The use of a ZBT is required since the data from the camera comes in first from odd lines and then from even lines rather than sequentially. Futhermore it must be held in memory to perform the necessary calculation for the next section without delaying the stream and losing data. The addresses are determined using the hcount and vcount of the data.

The data from the ZBT is accessed by the vram_display module. Like the ntsc2zbt module, the staff provided an initial version that red 4 pixels per address; however, the version used has been modified to read two pixel's worth of RGB data per address. While the option scaling the camera input in this to match the screen resolution was considered, it was rejected since the camera data is internal and not sent to the VGA and thus hidden from the users. The result is an 18 bit RGB pixel stream that is synchronized with hcount and vcount. However, since the addresses are determined by the hcount and vcount, there are numerous empty addresses that are read from the ZBT in the spaces where there is no matching camera data. This creates a problem since the ZBT is populated with random data in these locations. Therefore, before going to the video processing segment, the data is filtered once more. The pixels outside the camera window are turned black in this filter.

2.2 Video Processing

The video processing segment is responsible from converting the raw pixel stream from the camera input segment and using it to determine the location of the paddles, the power of swings, and the direction of the paddle movement. These calculations are made by a combination of three modules: cofm, velocity, and gamelocation. Figure 2.2 shows their interactions:

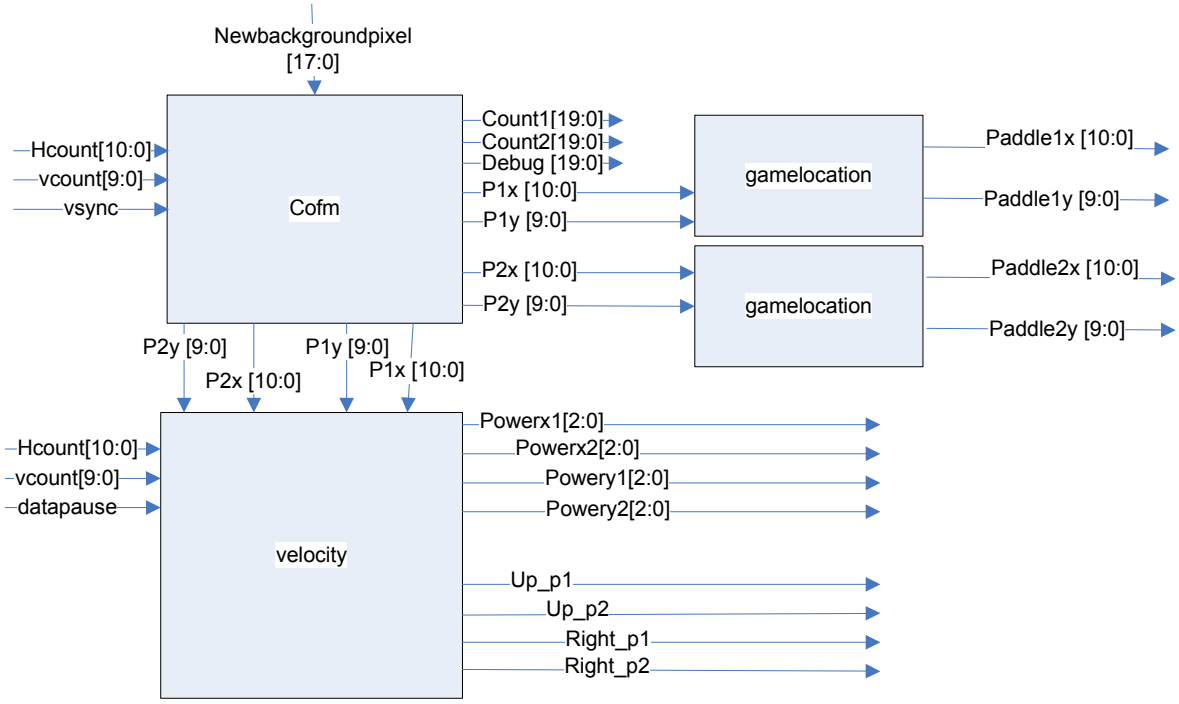

Figure 2.2 Video Processing Block Diagram

The first stage of the video processing segment is the center of mass module (cofm). This module takes in the pixel data along with hcount, and vcount and uses them to find the center of paddle1 (now white pixels) and paddle2 (now blue pixels). At each pixel, the module looks to see if the pixel is white. If it is, it increments the pixel count and then adds the hcount and vcount to two registers that stores the running sum of the x and y coordinates of the white pixels. It does the same thing for the blue pixels. Once the data finishes going through the camera window, it sends the counts and sums to a pipelined divider created using coregen. Since the game logic and display only need this data on frame changes, the new values of the location (p1x, p2x, p1y, p2y) are reset when vsync goes low signaling a new frame. At the same time, the sum registers and count registers are reset.

The important design element to notice with the cofm module is the timesharing of the divider module. Rather than creating several divider modules. Only one is used. Since the camera window ends at vcount = 500, the first of the four calculations is done at vcount = 501 and the last is done at vcount = 504. To accommodate latency, the sum and count values are constantly sent to the divider for the over thousand clock cycles per each vcount and then retrieved 500 cycles later (hcount=500).

The data from the cofm module is then sent to two modules, the gamelocation and the velocity modules. The gamelocation modules convert the location of the center of mass of each paddle in the camera window to the appropriate value for

the game. This is important for two reasons. First, the camera window is smaller than the full screen and second, the camera's x coordinates are inverted (when you move left the camera displays a movement to the right). The formula used to convert the center of mass is roughly:

Ygamelocation = 1.5*Ycofm

Xgamelocation = 1024- 1.5*Xcofm

Since the multiplication is by a factor of 1.5, rather than use a divider or multiplication module, the calculation occurs by adding Ycofm or Xcofm to itself shifted to the right one bit.

The final module in this section is the velocity module. This module is responsible for determining the speed and direction of the paddle. In order to compute these values, the module stores the last five frames' values of the two paddle's center of mass. To calculate the speed, it takes the difference of the current center of mass and the oldest stored center of mass. Since the range of possible differences is so large and spread out, the module uses the raw difference data to place the movement in one of six power buckets according to magnitude for both x and y movement.

The module also determines the direction by using the old data and the new data to determine if the paddle is moving up, down, left, or right. Like in the cofm module, the calculations occur on different hcounts and vcounts to limit the number of simultaneous calculations. Combined with the paddles' x and y powers, the result is a fairly accurate detection of swing vectors. The choice of using vectors rather than angles to calculate swings was made in an effort to eliminate any unnecessary timing delays.

2.3 Ball Speed

The ball speed section is unique from the other user input sections that exist in a vacuum from the gamelogic, the ball speed section is very heavily dependent on the game logic. It comprises of one module that is responsible for determining the speed the ball will travel on a hit based on the hitter, power, and direction. The figure below shows the various inputs and outputs to the module:

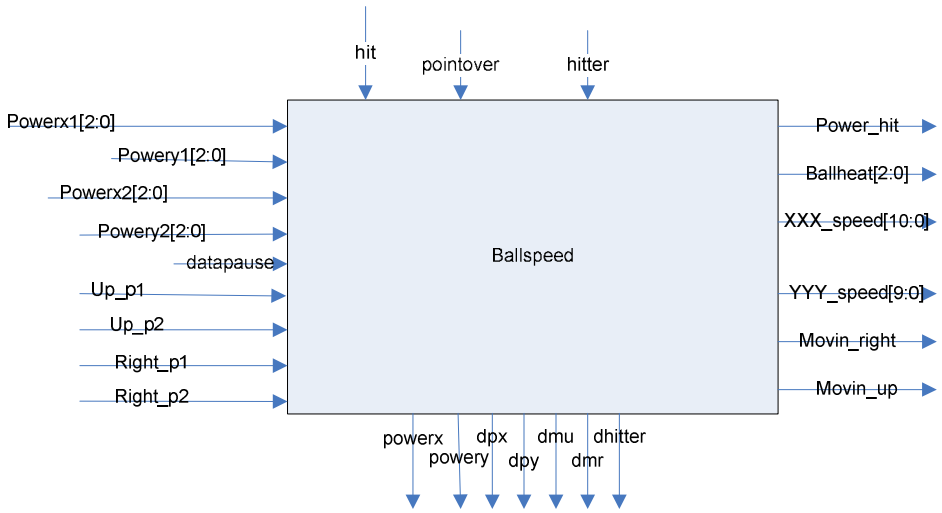

Figure 2.3 Ball Speed Block Diagram

This module works by taking in all the data computed by velocity and assigns powerx, powery, movin_up, moving_right based on the value of power and the direction of the current hitter. If the power is above a certain threshold (either power x is 5, power y is 5, or both powerx and powery are 4), the hitter is making a power hit and the ballheat increments. If power x and powery are below a threshold (both equal or less than 1), ball heat decreases by one. The purpose of the ballheat is to allow the game to speed up as rallies get longer. Using both the ball heat and the powers of the current hitter, the module sets a ball speed variable equal to the three times the sum of the power and the ballheat (note: xxx_ballspeed uses powerx and yyy_ballspeed uses powery). While the power might fluctuate between hits, the game logic only takes in the values on a hit so the fluctuations are acceptable.

The outputs on the bottom of the diagram (dpx, dpy, dmu, dmr, and dhitter) are outputs that are frozen on a datapause button press and hooked up to the hex display for debugging purposes.

2.4 Debugging the User Input Segments

The debugging of the User inputs was conducted differently for each segment. For the fist segment, the camera input, the debugging was primarily done using the logic analyzer and the VGA to test if the ZBT was being read and written into properly. This captured an error on the writing of data to the ZBT as a few registers were set to a wrong size. However, the largest effort for this segment was dedicated to calibrating the filter. Originally, filtering took place after the conversion to RGB; however, this led to numerous problems as the colors were too sensitive to light. This was corrected by moving the filter to the YCrCb space. In order to set the thresholds, a calibration module that tested various filters and tolerances was created. In addition, a round paddle was decided upon so that tilting the paddle did not drastically change the color.

The video processing module debugging and testing was broken into two sections, the cofm and game location module testing and the velocity testing. To test the cofm module, two methods were used, first attaching the counts and the sums to the hex display to see if the module was properly counting and second creating output blobs at the center of mass of each paddle. Figure 2.4 illustrates the blobs being drawn on the screen. The white represents player1 paddle and the represents player2's paddle. Each has a blob at its center of mass. The blue and red blobs correspond to the game locations respectively.

To test the velocity module, the debugging method was confined to using the hex display to show the speed, direction of swings. These were then used to determine appropriate buckets for the power.

The final section of debugging was the ballspeed module. For this, since tweaking inputs was important, the debugging occurred almost entirely of using writing test benches to simulate several combinations of inputs. Once integrated with the rest of the lab, the testing moved to using the hex display and buttons to freeze output values.

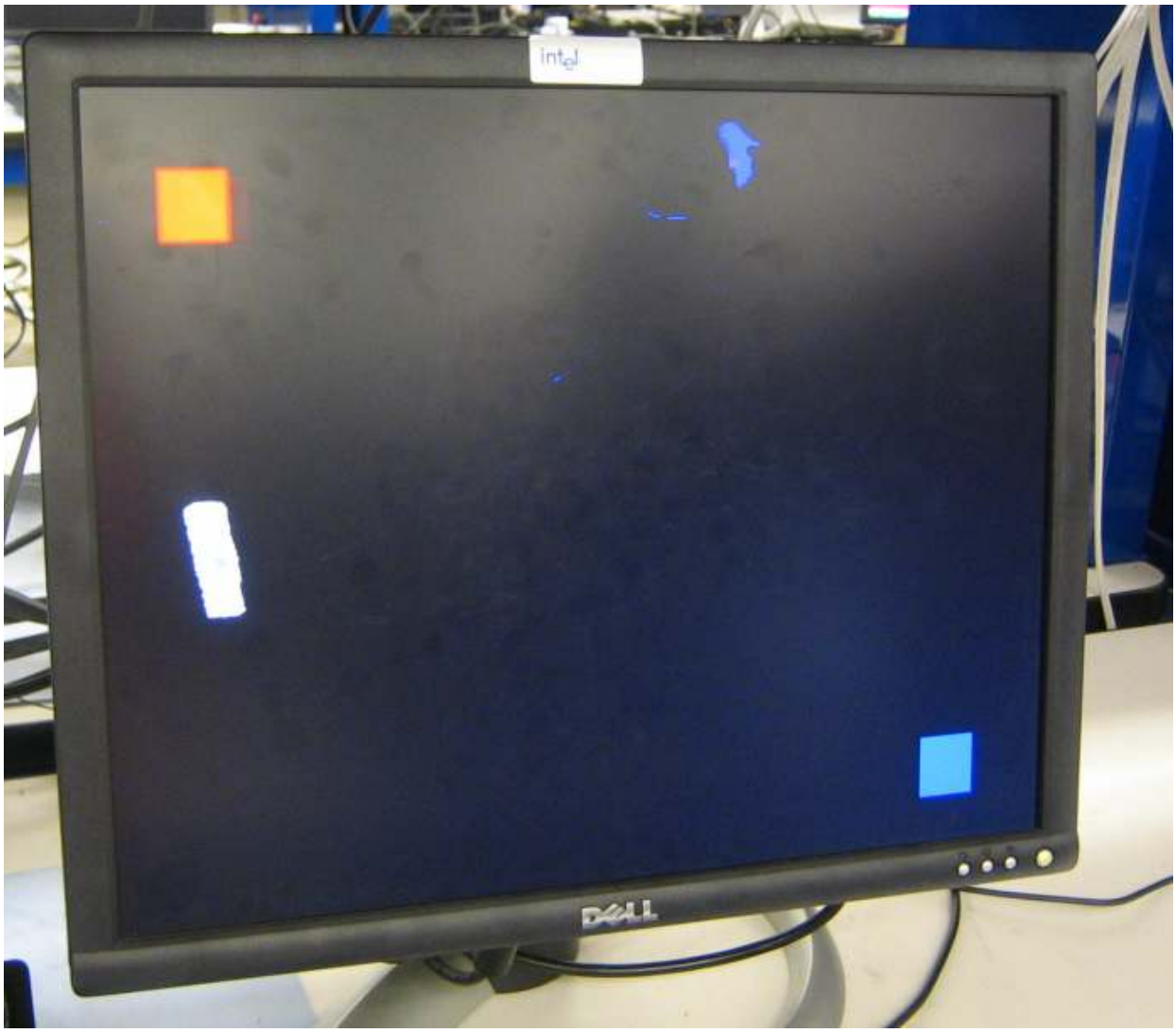

Figure 2.4 Filter and Cofm Debugging Screen

3. Game Logic
Azadeh Moini

Game Logic (GL) encompasses ball movement, collision detection, point over, and point tracker modules. GL takes inputs from the camera and input modules. It controls how

the ball moves on the screen, when the radius of the ball should change (important for 3-D movement), and when the ball should "bounce" off a wall. It outputs a signal when a paddle collides with the ball, and awards points to the server based on double-hits or misses. The GL keeps track of the rally score (hits per point), as well as each individual players' score. As soon as a player has fifteen points, the GL sends a signal to the output, indicating a screen-change. The four modules that comprise Game Logic are, to some extent, inter-dependent. Often, one module depends on the outputs of another module to operate correctly. All modules take the global reset and the 65 MHz clock as inputs, and all operate on the frame change (with the small exception of a delay in the ball movement module).

3.1 Ball Movement

The ball movement module began with a ball on the screen with nothing more than a changing radius. Depth and planar movement were then added, and after the ball was able to move continuously in a predetermined direction. Once the other modules started coming together, the ball could move according to various inputs and more details were added in terms of constraints.

In addition to the global inputs to Game Logic, this module takes as inputs x and y speeds (determined from the paddle motion), as well as whether the swing was up/down and right/left. It also takes the point over signal and the hit signal as inputs. Its outputs include the direction the ball is moving, and the location of the center of the ball, as well as the size of the ball radius.

This module operates on a four clock-cycle-delayed frame change. This was crucial to this module's functionality because of the hit input from the collision detection module. Before this change was added, this module operated off the frame change (as the other modules in GL do). However, at that rate, a hit in the collision detection module would not be detected in the ball movement module, and the appropriate change of direction would not take place. Delaying a few clock cycles ensures the high hit value is held long enough for the ball movement module to detect it and use it accordingly.

Upon reset, the ball movement module assigns the ball a location and radius within acceptable hitting range. The x and y speeds are held at zero; thus, the ball does not move initially. The same conditions apply to a point over. When a point is over, the ball is held at a different location than at reset. This was more for the purpose of debugging, so we could clearly see the difference between a reset and a point over and identify any errors in signals.

If the game is neither reset nor is the point over, the ball movement module goes on to check hit status and location of the ball. Much is determined by the radius of the ball. For instance, if the radius of the ball is small (one pixel or less), it means the ball is very far from the players and has hit either the back wall or the farthest corners of the side walls and should bounce back towards

the players. Otherwise, at a hit, the ball begins moving away from the players with speed and direction as indicated by the paddle swing.

The ball movement module also checks to see where the ball is along the walls. Rather than using a z-coordinate, I chose to simulate 3-D motion using only the x and y coordinates and the size of the radius. When the ball is moving away from the player, it is constantly decreasing in radius (opposite if moving toward players) and constantly (but in small increments) moving towards the bottom of the screen to indicate depth. Additionally, I essentially divided each wall (right and left) and the ground into several sections. If the ball is in one of these sections, it will bounce back based on the size of the radius. The inner portions of the walls and ground are farthest from the user, and the ball must be very small before it will bounce off from those locations. A similar (but opposite) statement is true for the outer portions of the walls. One of the challenges with this portion of the module was aligning the boundaries with the boundaries of the drawn court. Care was taken to include every pixel of the display in some boundary condition. Furthermore, the equations of the lines in the bottom left and bottom right corners were critical to accurate wall detection. The screen is set up as a normal coordinate plane, and equations of lines were manipulated to match the screen. Additionally, it was important to avoid negative numbers when calculating the equations of the lines. None of the variables are signed, and handling of negative numbers with unsigned variables is clearly unreliable. Therefore, the equations were manipulated and boundaries declared to avoid any negative numbers whatsoever.

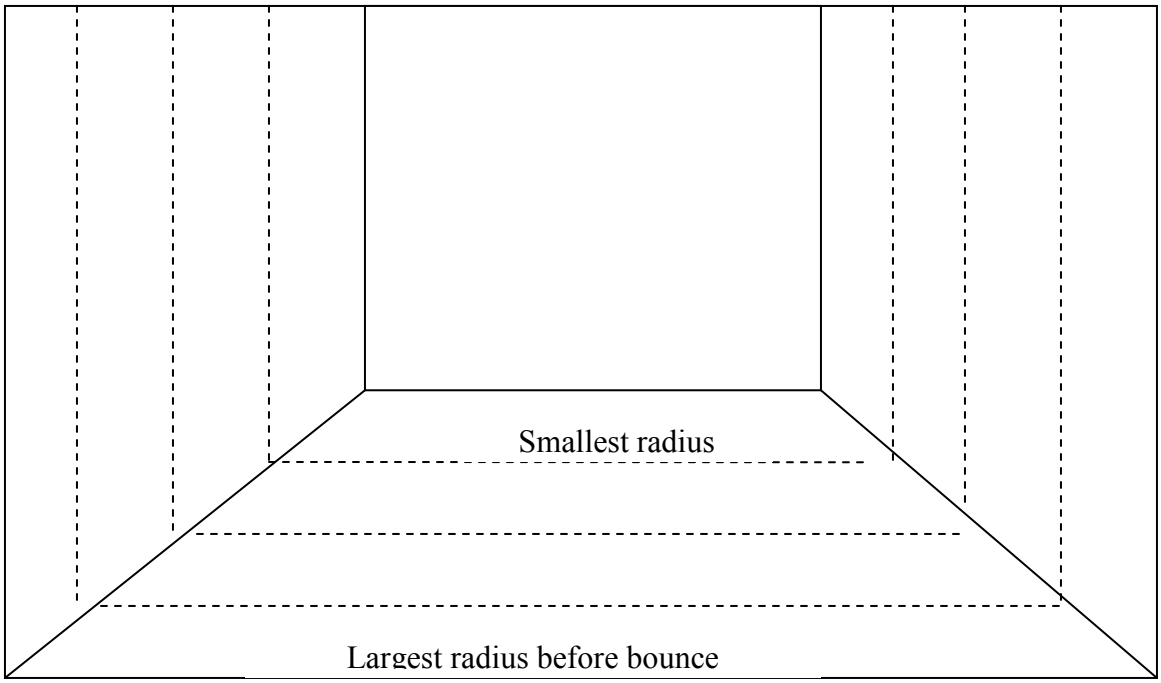

Figure 3.1: Wall/ground division

When in motion, the y-coordinate of the ball is constantly moving for 3-D simulation. However, there is also a y-speed element, as input from the paddle swing, that allows the ball to move in all planes in the court.

Furthermore, I added a one-bit counter to this module that changes its value every frame change. Before the x or y-speed is incremented, the counter must have a high value. Using this counter allowed me to slow the ball movement somewhat, as it will now only increment every two frames (rather than every frame). This is still faster than the player can detect, but avoids absurdly high ball speeds and outbursts of movement.

The ball movement module outputs the size of the radius, location of the center of the ball, and direction the ball is moving in to various other modules.

3.2 Collision Detection

The collision detection module is able to identify when a paddle and the ball overlap, which paddle is overlapping, and more specifically whether they overlap when the ball is both coming towards the player and the radius is within a specified hitting range.

This module inputs the radius, location of the ball, and ball direction from the ball movement module. Additionally, it uses the game and point over signals, the server signal, and paddle locations. The paddle locations come from the video input. The collision detection module sends two pulses: one when the ball is hit, and one when it is missed, and also indicates the current and previous hitters.

The reset values for the current and previous hitter are based on the server. They must both be reset to different values because a double-hit results either in the loss of a point or the loss of the serve; both cases are undesirable at reset. Because the first player to hit a ball at reset or after the loss of the point should be the server, the hitter is initially assigned to the non-serving player.

Before this module compares the ball's location to that of the paddles, it checks to make sure the ball is indeed coming towards the user and within the acceptable hitting range. If so, it compares the locations of the ball and paddles. It is conceivable that both players will have their paddles in the same location; thus, both players could potentially hit the ball. To make the hit less arbitrary and a bit clearer, precedence was given to the person who did not hit last. That is, if both paddles overlap, the hit is given to the person who, according to the last hit, should have been hitting this time.

Furthermore, a hit is not registered if the game over or point over signals are high. When game over is high, the court is no longer displayed on the screen; however, audio output continues, and clearly the game should not continue at this time until the game is reset. Additionally, when the point is over, a hit should not be registered. The ball resets its location at point over; if the non-server's paddle were to be at the reset location when the ball moves there, it will not move (recall that the ball is still while the point is over), but will still incorrectly register as a hit for the non-server. Not allowing a hit to be registered while the point-over signal is high avoids this problem.

The collision detection module is limited in that it only checks the center of the ball. For simplicities sake, a hit is registered only if the center of the ball is within the boundaries of the paddle.

The module continues to check the location of the paddles and ball until the ball is larger than the maximum threshold level. At this point, the ball has essentially moved too close to the player for it to be hit, and the ball is considered "missed."

3.3 Point Over

The point over module needs inputs from the collision detection module in order to operate correctly. These inputs include the missed ball signal and the current and previous hitters. Based on these inputs, it outputs whether the point is over, who the server is, whether the players' scores should be incremented, and whether a change in server has occurred.

As in the other modules, it is critical to reset point values and point over variable to zero at reset. Also, at reset, the server is set to zero, indicating that player one should serve. This is always the case for the beginning of the game.

To avoid problems with collision detection and ball movement, the point over signal was made to be held for only one frame length. Also, this is sufficient time for the ball movement module to input this information and reset the ball.

There are several cases in which the point is declared as over. One of these cases is the missed ball case. This was determined by collision detection and occurred when the ball was coming towards the player and its radius had exceeded hitting range. When this is the case, the point is declared as over, but point allocation is dependent on the server and hitter. The point over module analyzes two similar cases: one for player one serving, and one for player two. If player one is serving at the time of a missed ball, the module checks to see if the hitter was player one. Because the value held in the hitter register at the time of a missed ball would be the last person to hit it, if this is zero it is clear that the server was the last person to hit the ball. Thus, player two missed his opportunity to hit the ball and caused the point to be lost. Because player one was serving and player two erred, player one keeps the serve and earns a point. Had the last hitter been player two and player one missed the ball, no one would earn a point, but player one would lose the serve to player two. The case for player two's serve is similar.

The only other way to lose a point is on a double-hit. At every hit, the point over module compares the values of the current and old hitters. If both values are equal, one player hit the ball twice in a row, and resulting in the loss of a point. Determining whether a point was earned or whether the serve was lost, the module goes through a process very similar to that for a missed ball. If the server double-hit, he loses the serve to the other player. Otherwise, if the non-server double hit, the server earns a point and serves again.

When a point is declared as over, the new server output goes high. This signal remains high until a hit occurs. This was added for two reasons: a) it helped to have an audio signal at the beginning of a serve for ease in game play, and b) it is crucial in collision detection, and prevents the non-server from hitting the ball (and losing the point) on another player's turn to serve.

Because so much of the game logic depends on the value of the point over signal, it is important to set it to zero in any case other than the two described.

3.4 Point Tracker

As the name suggests, the point tracker module keeps a tab on the number of points each player has accumulated, as well as a running tally of the number of hits per point. Its main inputs come from the point over module and include the point over signal and whether player one or player two should get a point at any given time. In addition, it takes in the hit input. Its outputs are each players' scores, the rally score (number of hits per point), whether the game is over and a signal identifying the winner of a game. One additional

input to the point tracker module was intended to be a switch between one- and two-player mode. It is set to be held constantly in two player mode, but was set to one-player mode for testing purposes often. The one player mode does not keep track of player scores, only the rally.

The point tracker uses the reset button (as in the other modules) to reset the values of the player scores. The inputs it receives from the point over module dictate whether each player should receive a point or not. The point tracker module uses two registers to hold the value of these variables from the last frame. In this module, the register value is compared to the input value. A change in the value is indicated when the input is high (player x should get a point) and the old value is low. At this rising edge of the player point, the player's score can be incremented (one point at a time). If either player reaches fifteen total points, the game is declared as over, and a signal is output indicating the winner. This signal is held until the next game over signal. The game over signal itself is only a frame-length pulse. This is sufficiently long because the output screen does not change from the end screen back to the game until the reset button is hit.

The rally keeps track of the number of hits that occur within each point. It can be a larger number than the player score, although realistically it is unlikely that it will be too much larger. The point tracker module takes the hit input to calculate the hits per point. At a hit, this value increases. It resets at a new game, but also each time the point is over.

4. Outputs
Will Fotsch

I was responsible for designing the court and creating the different images that would appear on the screen including the ball, the two paddles, the court, and the scoreboard, which keeps track of the ball's current heat, the number of hits in the current rally, and the scores of both Player 1 and Player 2. I displayed all of the images on the screen using combinational logic as the hcount and vcount changed with the vclock as opposed to storing sprites into memory. Because of this, it took a great deal of effort to create an image on the screen and it was not as easy to change images as if I had used sprites from ROM that could be duplicated and placed anywhere on the screen. This implementation also involved a great deal of pipelining in order to avoid glitches on the screen. If too many calculations were taking place within one clock cycle, registers had to be added in order to pipeline the display. This also limited my ability to create complex images or use a wide variety of colors.

With the squash game we were creating, though, the images were sufficient to create an attractive display and allowed for easy changing of the ball's location and size based on the game physics and logic. I was also responsible for adding sound effects to the game in order to increase the entertainment value of the game. Eight BRAMs were created in order to handle eight distinct audio clips which could be programmed into the labkit

using a microphone and switches 5, 6, and 7. These sounds include a hit sound, a power hit sound, a player 1 serve sound, a player 2 serve sound, a player 1 terminated sound, a player 2 terminated sound, a new game sound, and a taunt sound.

Most of the work for the video and audio was done inside the large squash_game module. This module interacted directly with the recorder, inputs from Sumit's video processing, the recorder (handled audio), and the xvga in order to display pixels on the screen. The squash_game module created an output pixel, hsync, and vsync for the xvga. It also created a pulse signal play_pulse and a three bit signal called play that interacts with the recorder module in order to play audio clips from the FPGA.

4.1 Video

At the lowest level, all of the video display was created with modules similar to the blob module from lab 5. These modules create the appropriate shapes to be drawn to the screen based on hcount and vcount. On the upper levels is a series of instantiations of the appropriate modules in order to create the on screen displays, including the seven segment digital displays, the paddles, the ball, the words displayed, and the court. These pixels are chosen in order of layers so as to give a three dimensional perspective as opposed to combining the pixels with just or gates. This logic all takes place within the module squash_game. The paddles are the top pixel, which create hollow rectangles so that one can see the ball through them in order to hit it back. The ball layer is next so that it is bouncing on top of the court and over the scoreboard. The scoreboard goes over the court to display the appropriate rally numbers, player numbers, and heat bar with associated text. The court is the lowest layer as the background of the game. The end screen (Terman) takes precedence over any of the other pixels when the game is over so that if the game is over, the end screen is all that is displayed. Figure 4.1 shows the screen during game play.

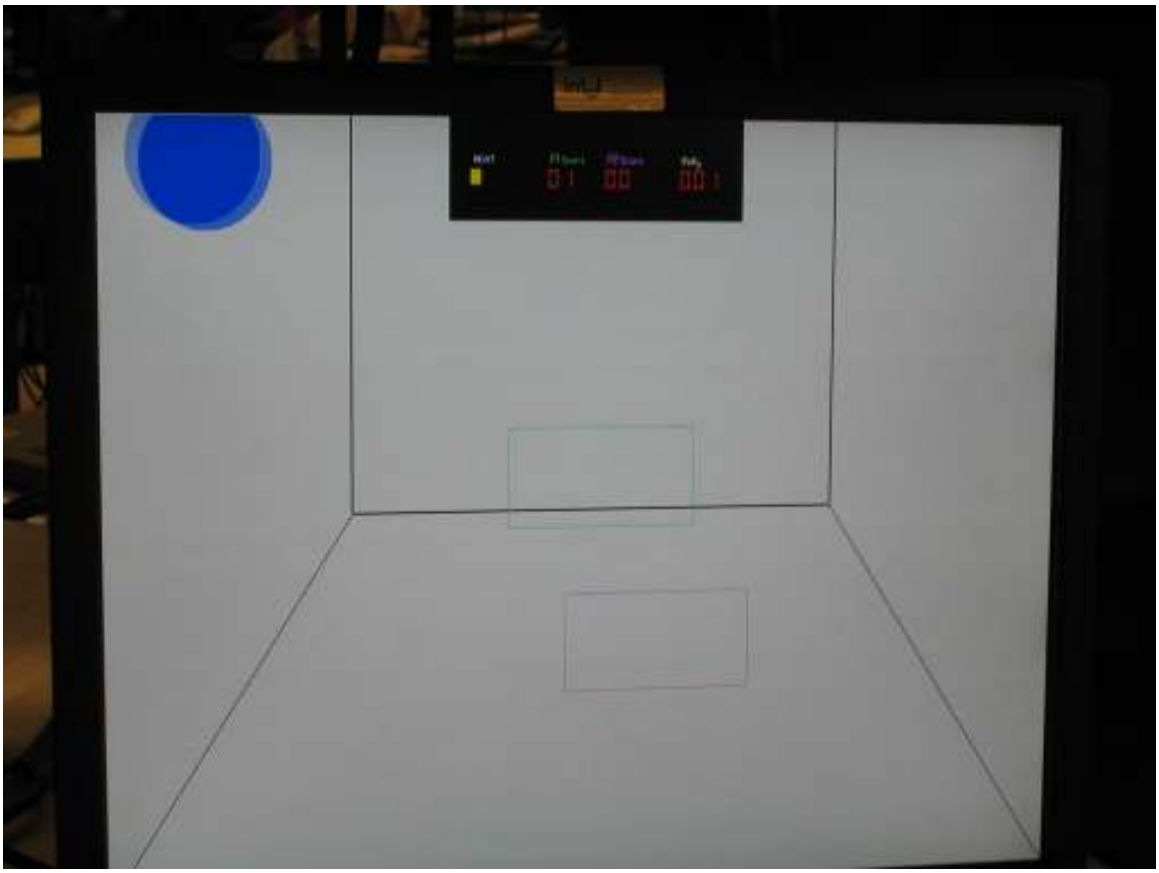

Figure 4.1: Display on Monitor During Game Play

4.1.1 Scoreboard

The Scoreboard is one of the largest display modules inside of squash_game. It is responsible for a number of different elements including the actual words drawn out, HEAT, P1 Score, P2 Score, and Rally. The scoreboard is also responsible for the dynamic displays, which include a bar for the heat of the ball, two seven segment displays each for player 1's score and player 2's score and three seven segment displays for the number of hits in the current rally. In this way, the scores are displayed as two digit decimal numbers and the rally is displayed as a three digit decimal number. The Scoreboard takes the inputs for player 1 and player 2 scores and breaks the numbers into decimal digits to send to the seven segment displays. For instance, if the number 15 was input as player 1's score, the Scoreboard would check and see that it is greater than or equal to ten and would display the first digit as one. It would then subtract ten from fifteen, leaving five as the second digit. Logic was created like this in order to handle any rally number from 0 to 199 and any player 1 or player 2 score from 0 to 19. These single digit decimal numbers are decomposed into the proper segments to display and sent to seven segment displays in the sevensigdisplay module. This functionality can be seen in Table 4.1.

The hits_per_point signal is an input which represents the number of hits in the current rally. It is a ten bit binary number; thus, it is inherently limited to 512. We decided that it is unreasonable to expect two players to be in a rally longer than 199 hits; thus, the current combinational logic for displaying the rally only supports up to 199. The game is over when a player gets 15 points, so there was no need to create logic for any greater than 19 points for the score of either player 1 or player 2. However, it would not be difficult to add the logic so that the rally score displayed up to 999 or the scores displayed up to 99. It would just require more logic in the form of if statements.

Originally, the heat bar was going to be another digital display much like the scores and rally, but we though it would be neat to have a bar increase in size to represent the ball's heat qualitatively rather than quantitatively. Thus, the heat bar is a rectangle that changes size based on the input of heat to the heat module.

Also in the scoreboard are the words HEAT, P1 Score, P2 Score, and Rally. These words were created with a series of blob modules (heat, player1score, player2score, and rally, respectively) in order to spell out the words on screen.

4.1.1.1 Seven Segment Display:

These displays were created with a series of blobs. The blobs are chosen based on logic based on the input number. A case statement chooses which blobs to write based on the input number in the Scoreboard module. Each seven segment display can handle any decimal or hex number from 0 to 15. Each seven segment display has lines two bits wide, with the width of the entire seven segment display being 10 pixels and the height of the entire seven segment display being 18 pixels. The basic structure of the seven segment displays can be seen in figure 4.2.

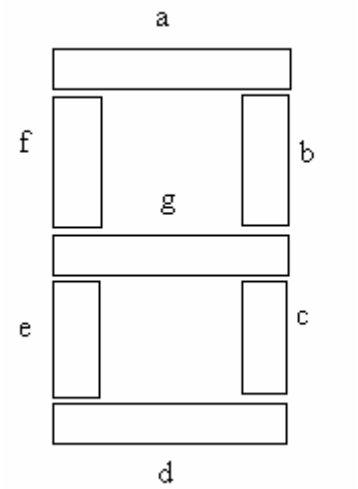

Figure 4.2: Seven Segment Display

Table 4.1: Numbers Corresponding to the Rectangles of the Seven Segment Display

hex/decimal numbers	Letter rectangles to be displayed
0	abcdef
1	bc
2	abdeg
3	abcd
4	bcfg
5	acdfg
6	acdefg
7	abc
8	abcdefg
9	abcdfg
A	abcefg
b	cdefg
c	deg
d	bcdeg
E	adefg
F	aefg

4.1.1.2 HEAT

The word HEAT is drawn in white on the scoreboard. It takes an upper left x,y coordinate and draws out the word "HEAT". It is drawn using a series of blob modules. The letter H is 7 pixels high and 4 pixels wide. The letter E is 7 pixels high and 3 pixels wide. The letter A is 7 pixels high and 3 pixels wide. The letter T is 7 pixels high and 5 pixels wide. The individual rectangles are put together pixel by pixel with or gates.

4.1.1.3 Player1score

The word "P1 Score" is drawn in green on the scoreboard. It is green to match player 1's paddle. It takes an upper left x,y coordinate to place P1 Score on the screen in the appropriate place according to the numbers input from the Scoreboard module. The letter P is 9 pixels high and 5 pixels wide. The number 1 is 9 pixels high and 1 pixel wide. The letter S is 9 pixels high and 5 pixels wide. The letter c is 5 pixels high and 3 pixels wide. The letter o is 5 pixels high and 4 pixels wide. The letter r is 5 pixels high and 3 pixels wide. The letter e is 5 pixels high and 3 pixels wide. The pixels for these rectangles are put together with or gates, just like HEAT.

4.1.1.4 Player2score

The word P2 Score is drawn in magenta in the same way that P1 Score accepts the upper left x,y coordinate. The 2 is 9 pixels high and 5 pixels wide and all the other characters are identical to P1 Score. Again, the rectangles are put together with or gates.

4.1.1.4 Rally

The word Rally is drawn in white the same way as the other words were drawn on the screen with blobs. The letter R is 7 pixels high and 4 pixels wide. The letter a is 5 pixels high and 3 pixels wide. The letters l are both 7 pixels high and 1 pixel wide. The letter y is 6 pixels high and 3 pixels wide. The pixel output of Rally is again the result of the different rectangles put together with or gates.

4.1.2 Paddles

Another upper level module within Scoreboard is the Paddle module. This module takes the x,y coordinates of both player 1's paddle and player 2's paddle. Paddles were created using the blobcheck module. Player 1's paddle was drawn in green and player 2's paddle was drawn in magenta. These colors were chosen in order to be easy to pick up over any part of the screen. Player 1's paddle matches the color of P1 Score and player 2's paddle matches the color of P2 Score so that it is very easy for a player of the game to realize which paddle is theirs and which player number they are without confusion. These two paddles are each rectangles 200 pixels wide and 100 pixels high. They were designed to be hollow rectangles in order to allow the user to see the scoreboard and the ball through the paddles, but still have the paddle clearly on top of the other layers in order to show that the paddles are between the user and the court and ball.

4.1.3 Court

Court uses a series of modules in order to draw the background squash court for the user to play on. The walls and floor were chosen to be white with black lines representing the corners of walls. The back wall was drawn with one large white rectangle with a small black rectangle over it to be the background of the scoreboard using instantiations of the blob module and blobinv module. On the left and right side of the screens are white rectangles 2 pixels away from the back wall in order to create the lines representing the corners also using instantiations of the blob module. The middle of the floor was drawn in the same way. The lower right and left corners are actually also rectangular shapes, but were created using the blobline and blobline2 modules. These modules were designed so that white pixels would be drawn everywhere except for the corners of the back wall to the lower corners of the screen. This gives the user a three dimensional court that displays the floor, side walls, and back wall. It is the perspective of an entire squash court from near the ceiling of the court.

4.1.4 Terman

The Terman Module was created in order to draw a crude pixel of Professor Terman's face with the word "TERMAN-ATED" displayed on screen. This screen is displayed at the end of a game. It draws Professor Terman's face as a rectangle with rectangular features and circular eyes. The eyes were created with

the ball modules and chosen to be blue. His skin is created with a large white rectangle using the blob module so that the other features could be drawn on top of the skin. He was drawn essentially having side burns and no hair with a mustache. His hair was represented with yellow rectangles drawn by instantiations of the blob module. His lips were drawn to be magenta in a rectangular fashion (blob module) and his nose was drawn with two thin black lines as the outline (instantiations of blobinv module). He was given green rectangular glasses as well (using blobcheck module). This screen displays from squash_game based on the game logic for when a game is over and will stay on screen until a user hits the reset button. The screen can be seen in Figure 4.3.

Figure 4.3: Game Over Screen, "TERMAN-ATED"

4.1.5 Lower-level Modules

These modules are the building blocks of the video display. Each of these modules accepts as input an x,y location to draw onto the screen, a color, and the vclock (65mhz), vsync, and hsync signals.

4.1.5.1 Blob
Blob is a module that is used for rectangles on the screen. On the positive edge of the video clock (65 mhz) it checks vcount and hcount and if the current vcount and hcount are within a set x width and y height, it will draw pixels of the appropriate color. Otherwise, it will leave blank pixels.

4.1.5.2 Blobsize
Blobsize is a module almost identical to blob, but it accepts width as an input instead of having width as a parameter. This is used for the heat bar display in the scoreboard. When heat changes, the width of the rectangle drawn changes appropriately

4.1.5.3 Blobinv
Blobinv is a module similar to blob, but when hcount and vcount lie outside of a certain x and y boundary, the output pixel is white instead of a blank pixel. In this way, black pixels can be drawn on top of white pixels. Blobinv should create output pixels that are white everywhere except within the boundaries of the x, y coordinates and the length and width specified. This way other modules can check to see if the pixel created by blobinv is zero and draw a black pixel in those places on the screen instead of a color pixel.

4.1.5.4 Blobcheck
Blobcheck is a module similar to blob, but it only draws pixels in on the border of the rectangle, 2 pixels wide. This is used for the paddles so that the paddles can be seen over top of the ball, but the user can still see the ball through the paddles. It is also used for the rims of the glasses in the Terman end screen.

4.1.5.5 Blobline
Blobline is a rectangle drawing module that draws a diagonal line through the rectangle created. It checks hcount and vcount to see if it is within certain x and y boundary conditions. If pixels are, then they are drawn to the screen unless the pixels fall on the black line with equation y=3x/2. Thus, this module creates rectangles with diagonal black lines running through them from upper left to lower right. This module was important to pipeline because of the multiplication and division that must take place in each clock cycle.

4.1.5.6 Blobline2
Blobline2 works like blobline, but with the diagonal line through the rectangle running across the opposite corners (lower left to upper right). The line's equation is y0-3x/2 = y where y0 is the y coordinate of the upper left corner of the rectangle. It checks x and y boundary conditions the same way as any of the other basic pixel modules.

4.1.5.7 Ball
The ball module is both an upper level module inside of squash_game as well as a basic building block module. The ball module works like the blob modules in

drawing pixels based on hcount and vcount being in the appropriate range of display. It uses basic algebra to calculate the pixels in which to draw based on the radius and x and y locations of the center of the ball ($r^2 = x^2 + y^2$). If hcount and vcount fall within the bounds of the circle, the ball module will draw pixels. With the extensive math here, this module was important to pipeline in order to avoid glitches in the output.

4.1.6 Synchronizing/Pipelining

In order to allow the screen to display properly with no memory and a series of modules creating pixels, pipelining was necessary. This also required delay of the hsync, vsync, and blank outputs of the squash_game module in order to match the pipelined propagation delay of squash_game's pixel output. The pipelining was done in order to minimize the number of calculations necessary in each video clock cycle (65Mhz). This was done in order to solve video output glitches, particularly with the ball module, but also involved when using a series of many or statements when combining the video display. The total delay from input hsync, vsync, and blank signals until the output pixel is displayed is 15 clock cycles. With a 65mhz clock, each clock cycle is approximately 15 ns. This means the total propagation delay is 225ns. This delay (1/4400000 s) is quick enough that as far as a user's eye can tell, the game is happening in real time.

4.2 Audio

Audio was done in much the same way as in Lab 4. Instead of 64kx8, the BRAMs were 32kx8 to store short sound clips. Eight of these were created to store 8 different audio clips. These audio clips can be recorded and programmed based on pressing and holding button two. Each clip should play properly based on inputs of hit, power_hit, newserve, server, gameover, reset, and buttonL. These variables will determine the output of squash_game, play. Play is a 3 bit output that can hold a number from 0-7 corresponding with the appropriate sound to be played. When the appropriate sound should play, squash_game sends a one frame length pulse to recorder in order to play an audio clip. When this pulse is high, the soundclip stored in the appropriate BRAM will play once. As long as the sound clip is long enough, the sound will be played exactly once. The frame length pulse is longer than one clock cycle of the audio's 27 mhz clock. When the soundclip is finished playing, no audio will be played until the next play_pulse signal from squash_game is sent to recorder.

The 8 different audio clips are completely programmable. This allows us to set original audio signals, but gives the user the freedom to reprogram audio clips as they see fit. Audio signals are programmed using switches 5-7 and button 2. When the user holds button 2 down, that is the signal to record. The user can hear what they speak in the microphones as they program a sound into the BRAMs. When button 2 is pressed down, playback is low meaning record mode. Then the recorder selects the appropriate BRAM using the switch values of Switch 7-5. It

creates a write enable signal for the proper BRAM to hold the current audio clip. This data is held in the BRAM until it is reprogrammed by another press of button 2 with the same switch values. See Table for list of sounds, the appropriate signals to play them, and the appropriate switches to store them into their respective BRAM.

Inside of squash_module is a series of logic to determine the appropriate value of play to be sent to the recorder. The circumstances of when audio clips play can be viewed in Table below. If it is player 1's turn to serve, the value of new_serve goes high and the logic inside squash_module checks to see the value of the server (you_serve). A pulse is created so that the recorder will play the appropriate sound, either "Player 1 Serve" or "Player 2 Serve." If a hit occurs, the logic checks to see that hit is high and sends a pulse with the appropriate value of play in order for the recorder to play the clip, "Thwack." A power hit takes precedence over a hit so that if a power hit occurs, power_hit is high and the value of play will correspond to the clip, "Power-up." If the reset button is pushed, that restarts the game and the clip "New Game!" is played by the recorder. If the game_over signal goes high, that means the game is over and the value of who_won will dictate whether the recorder plays "Player 1 Terman-ated" or "Player 2 Terman-ated." The last audio clip is based on the press of the left button on the FPGA. This will make play take the value that represents the audio clip, "You Suck!"

4.2.1 Recorder

The recorder module is the module used to generate and play audio clips using the FPGA's built in ac97 capabilities. Inside the recorder, each BRAM has as input from_ac97_data and a write enable signal that is based on switches 5-7 and pressing of button 2 to record. The output of the BRAM's are selected based on a case statement inside of an always block. When the pulse from squash_game is high, the recorder sets a variable, doneplaying, to zero to allow the recorder to play an audio clip. This doneplaying value will go high to one once the last used address space in the selected BRAM has been output in the form of an audio signal. The case statement selects the current value of the three bit play signal that comes from squash_game in order to play the appropriate sound according to the game's logic. The circumstances under which audio signals can be played, along with the appropriate values of the switches in recording can be seen below in Table .

Table 4.2: Audio Clips, Location, Switches to Program, and Signals to Play

8 BRAMs	Switch[7:5]	Play[2:0]	Situation under which it is played	Signal(s) in squash_game	Sound programmed
ram32x8	3'b000	3'b000	A power hit takes place	power_hit	"Power-up"
ram32x8_1	3'b001	3'b001	A regular hit takes place	hit, not power_hit	"Thwack"
ram32x8_2	3'b010	3'b010	Player 1 wins the game	game_over goes high, not who_won	"Player 2 Terman-ated"
ram32x8_3	3'b011	3'b011	The reset button is pushed for	Reset	"New Game!"

			a new game to start		
ram32x8_4	3'b100	3'b100	A point ends and it is Player 2's turn to serve	new_serve goes high, you_serve	"Player 2 Serve"
ram32x8_5	3'b101	3'b101	A point ends and it is Player 1's turn to serve	new_serve goes high, not you_serve	"Player 1 Serve
ram32x8_6	3'b110	3'b110	A player hits button left to taunt the other player	button_L	"You Suck!"
ram32x8_7	3'b111	3'b111	Player 2 wins the game	game_over goes high, who_won	"Player 1 Terman-ated"

Testing/Debugging

The biggest issue with testing and debugging was with displaying a screen without glitches. This was difficult because of the timing issues with attempting to display a screen in real time without using memory. Each of the modules had to be carefully pipelined. The way I tested and debugged was just to compile the code and look at the screen in order to see if anything looked like it had glitches. Since my video output was completely created before any integration with Azadeh's logic or Sumit's inputs, the time of compilation was not a major issue.

A big issue I ran into was forgetting about the phsync, pvsync and pblank signals. Since I was pipelining my output, this was creating a delay in which the pixels were displayed on screen. The entire screen was shifted to the left before I matched up the delays properly. Once I delayed the phsync, pvsync, and pblank signals properly, this was no longer an issue. I tested the functionality of my logic for the displays by using a series of counters that could display 0-15 for the P1 Score and P2 Score and 0-199 for the Rally display. In this way, I could see functionality without waiting for the proper inputs. I also used a counter to make the paddles move in a set way across the screen to see if the movement caused any timing issues or glitches in the screen.

Audio was much more difficult to test. First, I attempted to get basic functionality with just one audio clip based on a button press that would play once instead of repeating over and over again. Once I accomplished this with the doneplaying variable, I created 7 more BRAMs for a total of 8 BRAMs. I used a counter to change the value of the play signal and played the audio clips from the different BRAMs based on changing values of the counter. This worked well, but the problem when integrating was that if the same signal took place twice in a row, it would only play once. Thus, I had to create the pulse signal inside of squash_game and create a series of logic to create pulses with the appropriate play values based on the game's logic.

5. Integration and Debugging

Integration and debuggins was a huge part of our project. The video inputs and outputs could be tested and checked for full functionality on their own, but combining the proper inputs and checking the game's physics and logic posed major challenges. The game logic code was difficult to test while she was creating it, so much of the testing of the logic came at the very end through the integration processes. This was done through compiling the code and reprogramming the FPGA.

At first, we had a switch to choose between user input's output to see the paddles and how well his image filtering was working and the game's video output. This worked, but as we made more changes and additional features, the Xilinx ran into issues when compiling. Timing errors due to the length of wires interacting with the ZBT caused glitches with the input paddles and on certain compilations the screen was completely black with no output.

After testing several area constraints to no avail, since the user input's output display was unnecessary for user functionality, we commented it out in order to display only the actual game with no switch to choose between displays. This worked, but there were still timing errors which resulted in glitches causing a ghost-ing effect that made the paddles move rapidly and hard to control on the screen. To solve this we had to use the tool's multipass place and route tool. However, this greatly increased down compilation times.

The main testing of the logic was through repeated testing and playing of the game in order to see if the ball, scoreboard, end screen, and audio were behaving properly. There were timing issues matching up the audio signal since it operates on a 27mhz clock and the video operates on a 65mhz clock. These issues were solved by creating un unnecessarily long frame length pulse to ensure the audio would pick up the signal and play the proper audio clip. The game logic was tested to make sure player serves changed appropriately, points were awarded correctly, and the number of hits in a rally incremented properly.

6. Conclusion

6.1 Sumit Khatod

Going through the design process from start to finish taught me a number of useful skills. First of all it showed me how to pull together my knowledge base and use it create something. Through all the hitches along the way, I learned the value of rigorous testing of system and how to identify potential problems. However most importantly, I learned to be more resourceful in actively seeking out the answer to questions I did not know the answer to. This came into play very heavily when using some of the more advanced features of the labkit not covered in class such as tweaking area constraints.

Reflecting back on the project, there are few things I would do differently. First, I would probably have switched away from using a camera detector and uses an infared light so to further reduce noise. I would also have tried to create a third dimension of controls as the player can not move forward and back. The final thing I do differently if given more time is to learn more about the xillinx tools. Due to the timing delays associated with the FPGA's wiring, we were not able to add the function to change the paddle's size as that caused the ZBT timing delay associated with the internal wires to be too great.

6.2 Azadeh Moini

One of the most important things I think I took away from this lab was a sense of dedication and commitment to a long-term project. This project took an immense amount of planning from day one, and it was important to keep working on a regular basis to try to keep up. I found the deadlines provided by the course to be extremely helpful guidelines for our final project.

When integrating all three parts of the project, I realized that there were errors in my logic due to overlooked cases. I was reminded of the benefits of working in a small group. Working with team members was beneficial during debugging, but also during brainstorming. In both cases, attention to detail and a fresh point of view were extremely helpful. With a project as large as this one, it quickly became challenging to keep track of all the possibilities within the game, and having the project guidelines and other group members was definitely advantageous.

If I had more time on this project, there are two main features I would like to address. One major one is collision detection. As I mentioned, the current method for collision detection only checks the center of the ball against the paddles. I would have liked to detect a collision when any part of the paddle and ball collide. This could possible effect the angle at which the ball moves as well (in addition to the direction of the swing). I would have like to have charted the coordinates for the ball and paddles and compared them to determine collision. Alternatively, it may have been possible (although much more complex and challenging due to the layers in Will's display output) to determine collision based on overlapping pixels in the ball and paddles. Another change that may have been interesting to add would have been the effect of gravity on the movement of the ball.

6.3 Will Fotsch

I learned a number of things from this project. The importance of outlining distinctive roles in a group project became very obvious in addition to splitting up the tasks effectively. Divvying up tasks appropriately so that each person could work on his/her section of code independently was key in order to complete the project. Three people working on the same piece of code at the same time is a waste of effort.

I also learned the importance of clearly defining goals and collaborating with group members. The project does not work if each person's section works independently of the others. The project will only work if each person's section works not only independently, but also in conjunction. The timing of each signal and the proper number of bits in each piece of information that traveled from one module to another must be very precise. It is important to have overlap in people looking at each other's code because often it is much easier to find a problem if it is your first glance at a section of code rather than

looking at it for hours on end. Also, often times someone else will have a better idea on how to more elegantly solve a problem or create the desired functionality.

Given more time, the audio signals would have been interpolated to give a cleaner sound. This would be a very easy change to make. I probably would have started using sprites for video display in order to show more complex images and move them around the screen more easily. I could have had more realistic paddles and a more detailed back court. I would have made a blob of some sort to move along the side of one of the walls and/or the floor going along with the radius changing in order to give the user a better idea of depth perception on when the ball is within the hitting range. It could turn red when it is near the edge of the screen indicating that the ball is ready to be hit and could be another color as it travels away with the ball and comes back after the ball bounces off of the wall. I may have added a start screen and would have made the graphics nicer on the end screen.