6.111 FALL 2015

SNAPPA REFEREE

December 9, 2015

De Jesus, Juan
Orton, Matthew

Contents

INTRODUCTION

Snappa is a drinking game played by two teams of two that sit on opposite ends of a long rectangular table. The teams take turns throwing a die to score points, until 7 points are reached for a win. The die must travel in an arc and clear a loosely defined "low line" the four players agree on before the game. If the die bounces on the table and goes off the opponent's end of the table (exiting through the side of the table does not score a point) without being caught, the shooting team scores a point. Our project, Snappa Referee, not only aims to eliminate the difficulty of deciding whether the die passed the height threshold by tracking the movement of the die, but also enhances the experience with added visual effects, by allowing replays of any plays, and a scoreboard to keep track of points in this game.

To achieve these goals, we developed an FPGA-based motion-sensor referee. For our project, we will utilized a tennis ball and a Frisbee wrapped in colored paper instead of a die for motion-tracking simplicity. The ball tracking system is the most vital part of our project, as it compares the ball to its background to determine its position in space. Our Referee can then utilize this video input information to determine whether the ball traveled high enough to cross the predetermined low line. The integration of other modules allows for replays and user-inputs, further enhancing the gaming experience. These modules are explained in detail later in this report.

Stretch goals for this project revolve about being able to decide whether the die in a Snappa game actually went off the opponent's end of the table (point) or the side of the table (no point), This can be achieved by modifying the Referee module to consider where the ball falls, plus attaching motion-sensors on the edges of the table to eliminate contentious edge cases.

The goal of this report is to explain in-depth how these modules were designed and integrated so that the SNAPPA Referee would work successfully so that future developers could effectively utilize this document in order to mimic or further develop our prototype.

Chapter 1

Overview and High-level Architecture

When choosing a final project, we wanted modules that were closely tied to the skills we learned from our labs while also requiring us to break into new problems. In addition, we wanted to make sure the project stayed fun to work on, so we chose to design around a game to keep everything light hearted at a high level. Snappa is a good game for this project because it is visually simple for an FPGA to be able to extract the relevant visual information and because there are ways in which an impartial observer could improve the quality of the game. Figure 1.1 shows a high-level overview of how we planned to enact our Referee.

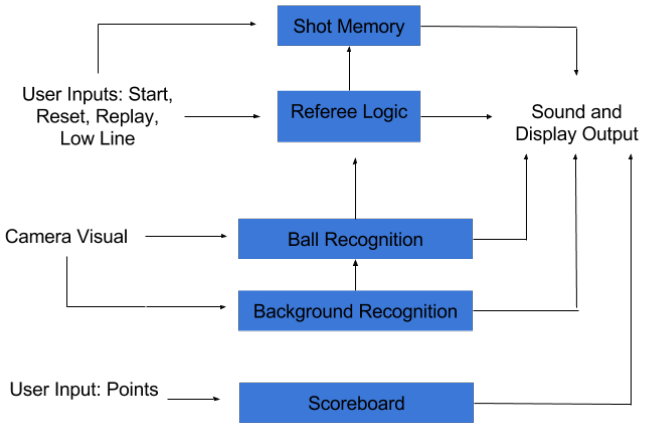

Figure 1.1: High level design of Snappa Referee

1.1 Design Choices

One of the primary concerns in designing a Snappa Referee was to make sure that our Referee would neither be a drag to use, nor take away from player's autonomy. We achieved this in a variety of ways. One of these was by allowing the players to be the ultimate judges of the validity of a throw by updating the scoreboard manually.

Another way in which we made the Referee a more informative guide was with a replay option. Reducing the system's interference with the original gameplay was most difficult when designing the replay module. We wanted the FPGA to know when to start and stop recording a shot with as little player input as possible. Ideally, there would be no required player input, but we realized it would be easy to confuse the system into thinking the next shot was starting very shortly after the previous shot had ended. While this could be worked around with a delay, we decided the system would be more robust if players provide a button push to signal the start of a shot and the FPGA determines the end of a shot. This does not alter the flow of the game much because offensive players must wait for the defense to be ready before throwing anyways.

1.2 Enhancing Gameplay

In a traditional game of snappa, the defensive team must call the shot low before the play has ended, so to simulate this, our system will show a red cross signifying a low throw once the ball is moving down after not crossing the low line, albeit no indication will be provided when the ball exits through the side of the table (our stretch goal). This is in part because our current Hue ball-tracking technology cannot discern effectively whether a ball exits through the end or side of the table unless our stretch goal is met of making corner-case detectors. Because the defensive player will often have a better perspective than a side-view camera as to where the ball left the table, this is not a major concern for our project.

The primary output for this system will be a display of the camera input with additional information. The threshold lines (Background Block) and the scoreboard will always be displayed on top of the background. After a throw, the display will indicate whether or not the shot was high enough, and if the system thinks the ball left the table from a side edge or the back edge. Additionally, the ball will be highlighted on the display so it can be more easily seen by the players during replays or while a throw is happening.

1.3 BASIC OPERATION

In this section, we will briefly explain how the flow of visual input would proceed to the modules, how it would be utilized, and how our project works from a high-level perspective. For simplicity, we've combined modules together into 'blocks' based on functionality. For reference, Figure 1.2 shows the physical set-up we used, which is simply an NTSC camera connected to an FPGA, and a table in front to play the game.

Figure 1.2: Physical Set up.

1.3.1 Background Block

This block keeps track of the spatial position of the threshold lines in order to give the necessary data to our FSM so that a decision about the validity of a shot can be made. Hence, this module takes in user inputs so that the lines can be moved. The threshold lines are as follows:

- Low Line: This is the line the thrown ball must cross in order to be a valid throw
- Table Top: This line signifies where the top of the table is in space. This will be utilized to determine when recording of a throw will end.
- Table Lines: These lines will be set to the edges of the table to provide the FPGA with boundary information on the table. This boundary information will be used to determine where a throw left the table, for our possible stretch goals.

1.3.2 Object Recognition Block

In this block, we are taking in visual input in the form of RGB values. After a conversion to HSV, the different hue values being input into the system by the camera will be processed in order to determine where the ball is located through a center-of-mass calculation. We have chosen to utilize this object due to its bright color and modest size (allowing for consistent motion-tracking). For demos from afar, a Frisbee proved to be even more consistent. Once the system has located the object, this module will keep track of it by storing its position as an x-y value from the camera input, continuously updating it based on where it sees the bright color of the ball again. This information will be placed on the display.

1.3.3 Shot Memory Block

This is the cornerstone of our project's ability to replay any throw. As explained previously, once the player pushes the "Start" button to begin a throw, the system will switch into the recording state, and begin storing video input (in the form of center-of-mass x and y locations) as Addresses, overriding the oldest video input once memory is filled. Recording only stops when our FSM switches from a recording to an idle state, signifying the end of a shot. Once a player wants to replay a throw, the Referee will call on this memory to be placed upon the display.

1.3.4 Referee Logic Block

This module is the arbiter of the project. It will take in the position of the ball in space, the background of the ball (the arena), plus the user inputs of Start, Reset and Line Positions in order to determine a result. This system has four states: idle, recording, replaying, and game over. State transitions as well as the function of each state will be covered in the implementation chapter.

1.3.5 Graphics Block

This module keeps track of points being input by the players. As explained in the design choice section, we chose to have the Snappa referee only as an unbiased guide, but the ultimate decision lies on the players themselves. This module gives information that will be displayed, plus is utilized to tell our Referee Logic when to transition into the game over state, once a team wins.

Chapter 2

Design and Implementation

In the Overview chapter, we mentioned the different components of our system (Object Recognition, Shot memory, Referee Logic, Graphics and Background display) separated by their different functions and how they came together. We will now explore the various modules that make up each of those components, and describe exactly how they were designed and implemented.

2.1 NTSC Camera to VGA Display (Matthew and Juan)

A variety of modules were given to us at the beginning of the project that would take in the ntsc visual input from the camera, and output it onto the display in black and white, in the form of YCrCb data. Modifications pertaining to address space were made to the ntsc_to_zbt module so that YCrCb values would be transformed into RGB, and properly stored in zbt for display purposes.

RGB values were then transformed using an RGB2HSV converter module that was provided, where no modifications needed to be made. Tracking, as explained in the Hue Detection module, is much more effective when done with HSV values.

2.2 Hue Detection (Matthew and Juan)

In order to track an object on camera, you have to process the camera image to determine what parts of it are desirable and which can be ignored. For this processing, we chose to use HSV(Hue, Saturation, Value) data from the image rather than the RGB data we display on the monitor. HSV data is extracted using a module provided to us, but this module is

computation intensive, so we had to delay all of our other video signals so they would line up with their corresponding calculated HSV.

We chose to do all this because Hue in particular is a more consistent image metric than RGB, so it allows the ranges we want to detect to be relatively stable. In order to test our ability to detect their hue ranges, We used four round objects, each a different color, and toggled between the positive hue ranges with a couple switches on our labkit. Pixels that matched the selected Hue range were colored magenta to distinguish them on screen.

In testing, we found that we narrowed the Hue ranges as much as possible while still allowing us to match the majority of each test object, but doing this was not sufficient to eliminate the massive amount of noise generated by background lighting and the camera itself. To cope with this, we decided to add additional range checking on saturation and value as well.

Ultimately, pixels were colored magenta if they matched Hue, Saturation, and Value, they were colored blue if they matched Hue and Saturation but not Value, and they were colored green if they matched Hue but not Saturation. Pixels that did not match the Hue range were either untouched or colored black depending on a switch on the labkit. Please see Figure 2.1 for reference.

Figure 2.1: VGA Display with blacked out camera image except for pixels matching combinations of HSV.

2.3 CENTER OF MASS (MATTHEW)

Once the pixel data from the camera is marked according to its HSV parameters, an average location has to be determined from all of pixels. This average, or center of mass, is found by taking the x and y coordinates (hcount and vcount) of every pixel in a frame that is a match, and summing them with their respective coordinates of the other matched pixels. These sums are divided by the total number of matching pixels in a given frame to produce coordinates for the center of mass of the frame. This center of mass position was displayed as a green pixel, as seen in Figure 2.2.

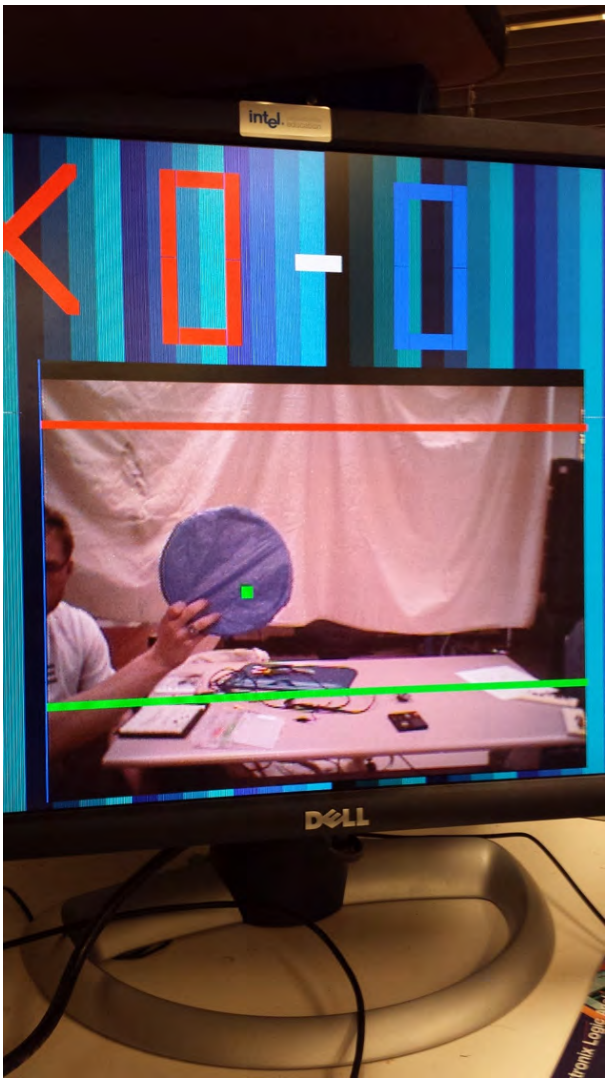

Figure 2.2: Green pixel shows the effective tracking of the object

It was a requirement for our project that the center of mass calculator and hue detector were both able to adjust to the rapid movement of a thrown object, but also be highly resistant to noise introduced by having the object relatively far from the camera. We tried a few different approaches, but we found the most success with limiting the window looked at for hue detection. When the designated switch was toggled, the hue detector module would only look at a fixed window surrounding the last calculated center of mass value instead of the entire camera image. With this feature, the center of mass calculator is highly resistant to noise because most of the image is ignored, and reacts quickly to movement as long as the object does not move entirely out of the window in the time span of one frame (0.05seconds).This tracking window can be seen in action on Figure 2.3.

Figure 2.3: VGA Display with blacked out camera image and matched values only around COM point

2.4 FINITE STATE MACHINE (JUAN)

As explained in the Overview Chapter, the FSM module is in charge of all game logic and hence, is the only module pertaining to the Referee Logic Block. Figure 2.4 shows the relevant inputs to the FSM. The FSM then outputs scores and whether a throw counts as a point for the Graphics module to display, plus its own state, to tell the Shot Memory module when to begin storing and deleting memory. The states, as well as their functions and transitions, are

explained below:

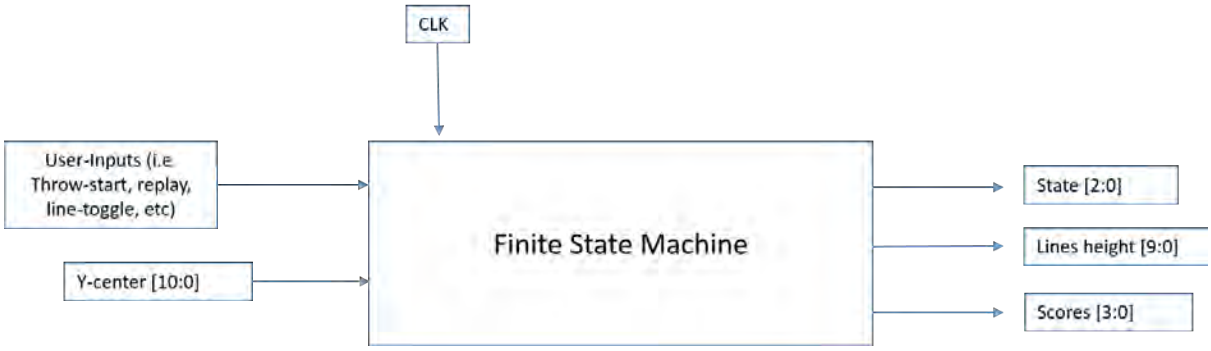

Figure 2.4: Game Logic FSM.

2.4.1 Idle

The FSM starts in the idle state once the game is first turned on. Pressing the reset button will clear all states and variables of the module and return to the idle state as well.

In the idle state, the system is waiting for any of a few user inputs. The threshold lines and the score can be adjusted using their designated button inputs in this state. The system will transition to the replay state if the replay switch is flipped, to the recording state if the button is pressed signifying the start of a shot and to the game-over state should a score reach 7 points. Logic was added so that a win by 2 is required as well.

2.4.2 Replay

In the replay state, the system will loop through the recorded video clip of the previous shot. The speed of this replay can be modified with push buttons, and the system will return to the idle state by flipping the replay switch back to its original position. More information about how to enact the actual storage of information in the Shot Replay module.

2.4.3 Recording

In the recording state is entered, the system will begin overwriting the memory holding the previous shot by utilizing our Shot Replay module, which begins recording when the FSM reaches this state.

The shot, and hence the recording, can end by two different mechanisms. The first one is when the center of mass of the object crosses the table line. Once this occurs, the system will

record for 2 more seconds, after which it will time out and transition back to idle. Whether the object passed the threshold height or not does not influence the transition, since that is simply used by the Graphics module to either output a checkmark, or a cross.

The other way a shot can end is if the object crosses the threshold line, but not the table line. This signifies that either the object was too high (leaving field of vision), or that the opposing player caught it before it hit the table (no point then). It transitions back to idle through a similar time-out mechanism as above.

2.4.4 Game-over

Once one side reaches 7 points (win by 2), the system will enter the game over state from the idle state. In this state a new screen will be displayed showing the final score and commemorating the winning side, as seen in the Graphics section. From here, the only transition is for the user to reset for the next game.

2.5 SHOT REPLAY (JUAN)

This module creates an array of registers where we can store center of mass values. There are 2^7 registers, each 25 bits long (so that 13 bits of x-center and 12 bits of y-center information can be stored). This array of registers (roughly 130 of them) allows for 130 different COM values. As we obtain one value per frame, and we have 20 frames per second, we can store 6.5 seconds of information, which we believe would convey all relevant information of the shot, thanks to our short time-outs.

This module is very similar to the mybram module explored in Lab 5. As given, this module would be outputting information, until a write-enable signal told the module to store information instead. For our purposes, it was modified so that it could write into memory (when state became ”record”), it could output from memory (when state became ”replay”) but it could also just stay idle, for when the system was in the ”idle” state.

Additionally, code was written to create a clock and an address increase only when a new center-of-mass calculation was outputted. The address number would get cleared when the FSM transitioned (so that each replay would start at the beginning, and each recording would be done on a fresh memory), and as a stretch goal, a button could be pressed to slow down the speed of the address variable so that the replay could be seen in slow motion.

Replay would send out its COM values to the graphics, which would then superimpose a pixel onto the display, as shown in Figure 2.5:

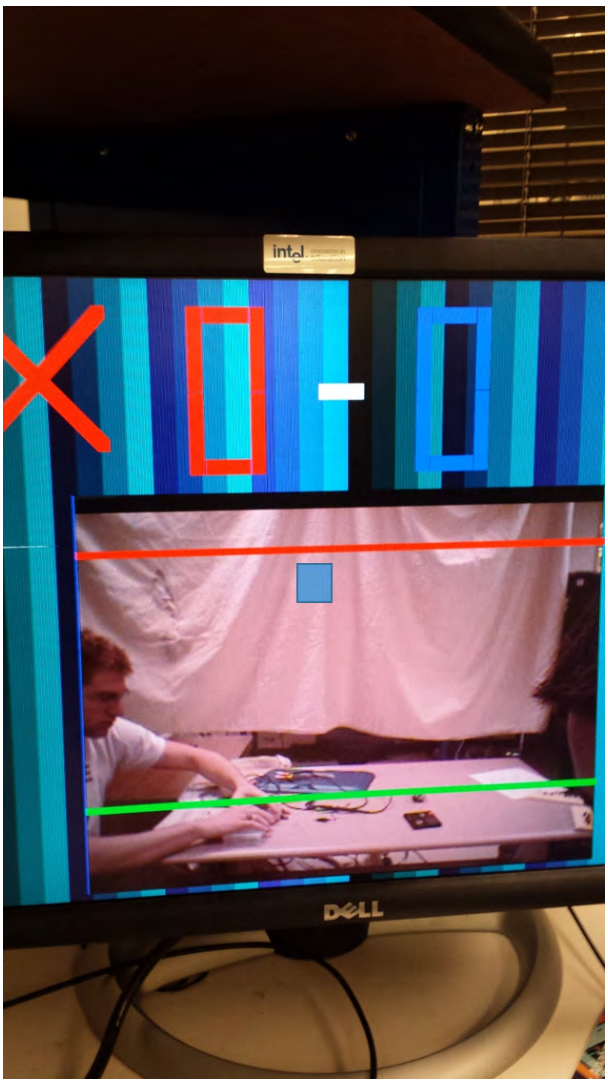

Figure 2.5: Replay in action.

2.6 GRAPHICS (MATTHEW)

The graphics we added to enhance the system are for the most part informative. The exception to this is the pattern of blue bars that serve as the background for the display. They were included in a module provided to us, but we chose to keep them for the final display because we found them visually appealing. All of the other graphic additions are variations on a simple sprite module that changes the value of a pixel to be displayed on screen for determined ranges of hcount and vcount.

2.6.1 Ball Marker

This sprite receives the coordinates for the calculated center of mass and displays a small square at that location. We use this marker to clearly see where the system thinks the ball is on the screen. If the system is in replay mode, the square is twice as large and a different color to more closely resemble the ball we are tracking rather than just a marker.

2.6.2 Threshold Lines

There are two lines generated on screen. The red line shows the threshold that a throw must exceed in order to be eligible to score a point. The green line marks where the table begins, so the system can determine once a shot has hit the table or dropped below it. The sprite that is used to generate each of these lines takes a height input from the FSM corresponding to a vcount and colors all pixels at that vcount within the area that the camera image is being displayed. The lines can be adjusted with buttons on the labkit that modify the height inputs provided by the FSM. These lines can be seen in Figure 2.6.

2.6.3 Low Indicator

The red ”X” on the VGA display is an indication that the previous throw did not successfully cross the low threshold line. If the previous throw had been high enough, a green check mark of equivalent size would be displayed instead, as shown on Figure 2.7. The shapes were made by writing out equations for the lines, rearranging them to remove all subtraction to avoid potential negative numbers, and then turning them into inequalities with an additional parameter for line thickness.

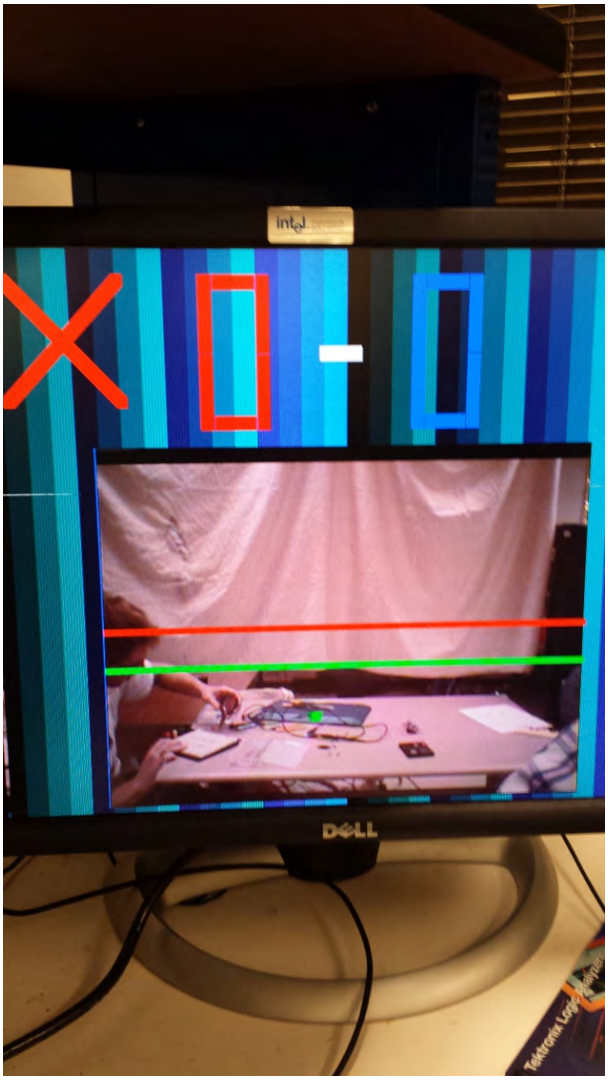

Figure 2.6: Threshold line in red, Table line in green

2.6.4 Scoreboard

The scoreboard contains three instances of a sprite module that acts as a segmented display. The middle of the three modules is fixed as a dash, but the other two receive score inputs that are passed along from the FSM. The score input determines which of the segments are turned on so the module displays the number corresponding to the score.

Figure 2.7: Checkmark appears when object is high enough

2.6.5 Game Over

Finally, when one team wins text is displayed on the screen indicating which team was the victor. The text is a single module containing modules for each letter as an individual sprite. The bounds for letters were made using the same approach as the low indicator. There is one input telling the module that the game has ended and another to tell it which side won. Figure 2.8 shows this game-over screen.

Figure 2.8: Final VGA Display

Chapter 3

Testing

Development of our project followed a pretty smooth path, with modules working in roughly the order we needed, until we had a final, integrated, working project. In this chapter, we will present how this process occurred from start to end, including any complications and how we went about solving them.

3.1 VGA DISPLAY AND DELAYS

Getting the camera visual input to display in color onto the monitor was not very difficult, and it had few set-backs. By the first week, we had a system where we were consistently outputting visual input, which was key to troubleshooting the rest of the project. We did encounter an issue with slight delays from the conversion to RGB, but this was quickly fixed using the delay modules provided to delay hcount and vcount by a few clock cycles.

3.2 HUE DETECTION AND CENTER-OF-MASS

Reliably matching desired pixels was a long process. We started by getting a sense of the Hue values of the objects through online resources. We refined these values by making blacking out all pixels, except those matched. This process is explained in-depth in the previous section on the Hue Matching module. These values were constantly refined throughout the project after every compiling. One problem we encountered was the fact that depending on the time of day, our system would work either very well, or not so well (due to incoming light from the window). We also tried out a variety of colors, and found blue and green to work exceptionally well. Red seemed to match the environment excessively, as shown in Figure 3.1.

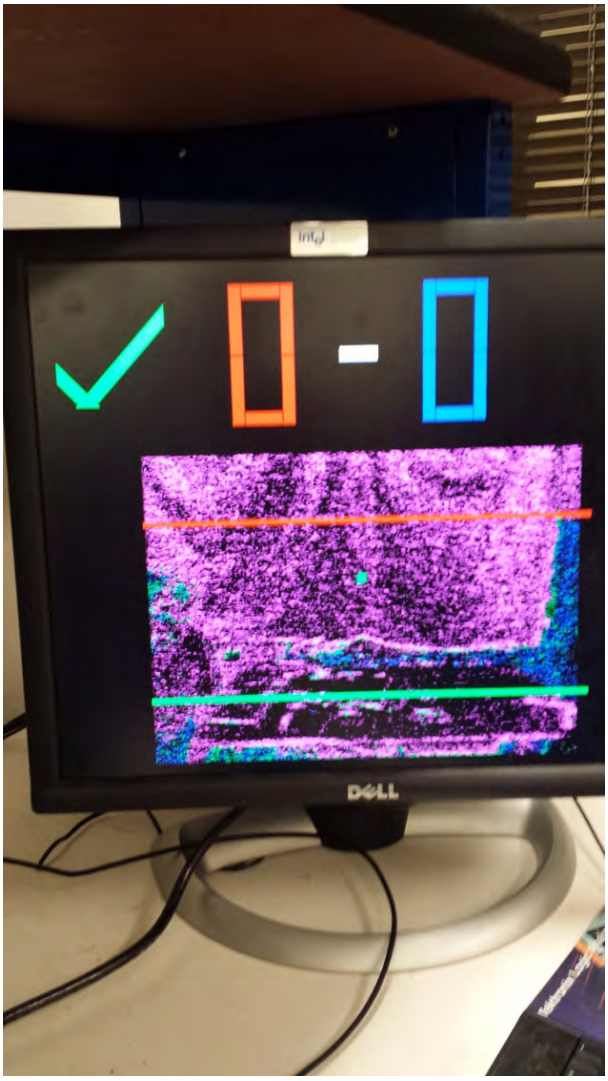

Figure 3.1: Red hue matching had too much background noise

This was the moment where we decided to use Saturation and Value to aid matching as well as using a window for matching to eliminate all ”noise”. Once we took these vital steps, we found the COM calculator to be working very well. Again, due to the amount of calculation involved in COM calculations, we settled on a delay of 60 clock cycles for all of the ntsc variables (hcount, vcount, etc).

3.3 GAME LOGIC AND REPLAY

Getting the FSM up and running went very well. We had written code for it early on, while still trying to get correct Hue matching and object tracking, so integration took less than a

day. We found that the FSM transitioned smoothly and correctly. We utilized the hex display on the labkit in order to be sure that the FSM transitioned as desired.

Replay code was also written ahead of time, but it took more time to successfully integrate. This was simply because we had to be very careful to clear memory at specific state transitions, plus keep track of addresses throughout the process. Overall, it only took a couple of days to enact a proper replay, including the option that allowed the replay to slow down to half of its speed (by cutting address increment and clock transitions to half the frequency).

3.4 Graphics and Game-Play

All graphics, as explained in the Graphics module section, were done as blobs, with a good bit of math involved to determine good shapes. On every compile, we took the opportunity to see how the blobs looked, and how they could be refined to look more appropriate. However, the final week was enough time to make sure all graphics were suitable, and transitioned based on the FSM transitioning.

We also had in mind, as a stretch goal, to add sound effects. We created a Coregen ROM that stored the necessary bit values, and then we utilized the code from lab 5 to output AC97 sound. However, we were unable to make this work correctly. Though the addressing into the CoreGen incremented correctly (as seen in the Hex Display), the CoreGen was not outputting any values. After a brief discussion with Miren, we decided there was little to do but drop it.

3.5 Conclusion

Our Snappa Referee provides a clean interface that enhances a favorite pastime of ours. It uses image data from a camera to track a thrown ball and determine whether the shot was high enough to be eligible to score a point. This effectively ends the greatest point of contention in what some might call the greatest game. Most importantly, it provided us with a very important experience experience: trying to program a real world system with real world constraints. We spent many hours combating noise among other enemies to bring forward a robust system, and we hope the work we have done will help those in years to come.

Chapter 4

Appendix

4.1 VERILOG

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is
used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time
for
// data to be stored to the ZBT during other pixel times.  The NTSC
decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//
// switch[7] selects between display of NTSC video and test bars
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//           during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)
//
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09
//
// Use ramclock module to deskew clocks;  GPH
// To change display from 1024*787 to 800*600, use clock_40mhz and change
// accordingly. Verilog ntsc2zbt.v will also need changes to change
resolution.
//
// Date   : 10-Nov-11

//////////////////////////////////////////////////////////////////////////
//////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////
//////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
```

```
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is
an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated
into
//    the data bus, and the byte write enables have been combined into
the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
////////////////////////////////////////////////////////////////////////
//////
//
// Complete change history (including bug fixes)
//
// 2011-Nov-10: Changed resolution to 1024 * 768.
//                            Added back ramclok to deskew RAM clock
//
// 2009-May-11: Fixed memory management bug by 8 clock cycle forecast.
//              Changed resolution to  800 * 600.
//              Reduced clock speed to 40MHz.
//              Disconnected zbt_6111's ram_clk signal.
//              Added ramclock to control RAM.
//              Added notes about ram1 default values.
//              Commented out clock_feedback_out assignment.
//              Removed delayN modules because ZBT's latency has no more
effect.
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb
devices
//              actually populated on the boards. (The boards support up
to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a
default
//              value. (Previous versions of this file declared this port
to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb
devices
//              actually populated on the boards. (The boards support up
to
```

```
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
////////////////////////////////////////////////////////////////////////
//////

module zbt_6111_sample(beep, audio_reset_b,
                  ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock,

             vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
             vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
             vga_out_vsync,

             tv_out_ycrcb, tv_out_reset_b, tv_out_clock,
tv_out_i2c_clock,
             tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
             tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

             tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
             tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
             tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
             tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

             ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
             ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

             ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
             ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

             clock_feedback_out, clock_feedback_in,

             flash_data, flash_address, flash_ce_b, flash_oe_b,
flash_we_b,
             flash_reset_b, flash_sts, flash_byte_b,

             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

             mouse_clock, mouse_data, keyboard_clock, keyboard_data,

             clock_27mhz, clock1, clock2,

             disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
             disp_reset_b, disp_data_in,

             button0, button1, button2, button3, button_enter,
button_right,
             button_left, button_down, button_up,

             switch,

             led,

             user1, user2, user3, user4,

             daughtercard,

             systemace_data, systemace_address, systemace_ce_b,
```

```
            systemace_we_b, systemace_oe_b, systemace_irq,
systemace_mpbrdy,

            analyzer1_data, analyzer1_clock,
            analyzer2_data, analyzer2_clock,
            analyzer3_data, analyzer3_clock,
            analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
        vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
tv_out_i2c_data,
        tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
        tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
tv_in_aef,
        tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
        tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
   output [18:0] ram0_address;
   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
ram0_we_b;
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b,
flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;
```

```
   input  button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
            analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock,
analyzer4_clock;


//////////////////////////////////////////////////////////////////////
///
   //
   // I/O Assignments
   //

//////////////////////////////////////////////////////////////////////
///

   // Audio Input and Output
   assign beep= 1'b0;
//   assign audio_reset_b = 1'b0;
//   assign ac97_synch = 1'b0;
//   assign ac97_sdata_out = 1'b0;
/*
*/
   // ac97_sdata_in is an input

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
```

```
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

/* change lines below to enable ZBT RAM bank0 */

/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;// clock enable
*/

/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

/**********/

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;

   //These values has to be set to 0 like ram0 if ram1 is used.
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

   // clock_feedback_out will be assigned by ramclock
   // assign clock_feedback_out = 1'b0;  //2011-Nov-10
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are
inputs
```

```
   // LED Displays
/*
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
*/
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;


//////////////////////////////////////////////////////////////////////
///
   // Demonstration of ZBT RAM as video memory

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

//   wire clk = clock_65mhz;  // gph 2011-Nov-10
```

```
/*
//////////////////////////////////////////////////////////////////////////
///
   // Demonstration of ZBT RAM as video memory

   // use FPGA's digital clock manager to produce a
   // 40MHz clock (actually 40.5MHz)
   wire clock_40mhz_unbuf,clock_40mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 2
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

   wire clk = clock_40mhz;
*/
      wire locked;
      //assign clock_feedback_out = 0; // gph 2011-Nov-10

   ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),
                           .ram0_clock(ram0_clk),
                           //.ram1_clock(ram1_clk),   //uncomment if
ram1 is used
                           .clock_feedback_in(clock_feedback_in),
                           .clock_feedback_out(clock_feedback_out),
.locked(locked));


   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clk, ~button_enter, user_reset);
      debounce db2(reset, clk, ~button0, button0_clean);
      debounce db3(reset, clk, ~button1, button1_clean);
      debounce db4(reset, clk, ~button3, button3_clean);
      debounce db5(reset, clk, ~button_up, button_up_clean);
      debounce db6(reset, clk, ~button_down, button_down_clean);
      debounce db7(reset, clk, ~button_left, button_left_clean);
      debounce db8(reset, clk, ~button_right, button_right_clean);
   assign reset = user_reset | power_on_reset;

   // display module for debugging

   reg [63:0] dispdata;
   display_16hex hexdisp1(reset, clk, dispdata,
                  disp_blank, disp_clock, disp_rs, disp_ce_b,
                  disp_reset_b, disp_data_out);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);
```

```
   // wire up to ZBT ram

   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;

   wire ram0_clk_not_used;
   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
              vram_write_data, vram_read_data,
              ram0_clk_not_used,   //to get good timing, don't connect
ram_clk to zbt_6111
              ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   // ADV7185 NTSC decoder interface code
   // adv7185 initialization module
   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                  .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                  .tv_in_i2c_clock(tv_in_i2c_clock),
                  .tv_in_i2c_data(tv_in_i2c_data));

   wire [29:0] ycrcb;  // video data (luminance, chrominance)
   wire [2:0] fvh;     // sync for field, vertical, horizontal
   wire       dv;      // data valid

   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                  .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                  .ycrcb(ycrcb), .f(fvh[2]),
                  .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

      // Get 24 bit RGB data out of 30 bit ycrcb data and reduce it to 18
bit RGB data
      wire [17:0] rgb;
      wire[7:0] R,G,B;

      YCrCb2RGB ycrcb2rgb (.R(R), .G(G), .B(B), .clk(tv_in_line_clock1),
.rst(reset),
                           .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]),
.Cb(ycrcb[9:0]));
      assign rgb = {R[7:2],G[7:2],B[7:2]};


   // code to write NTSC data to video memory
   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, rgb,//ycrcb[29:22],
               ntsc_addr, ntsc_data, ntsc_we, 1'b0);

   // code to write pattern to ZBT memory (The blue bars in the
background)
   reg [31:0]    count;
   always @(posedge clk) count <= reset ? 0 : count + 1;

   wire [18:0]   vram_addr2 = count[0+18:0];
      // Select two possible widths of blue bars for the background
   wire [35:0]   vpat = {4{count[3+4:4],4'b0}};
```

```
   // mux selecting read/write to memory based on which write-enable is
chosen
   wire    sw_ntsc = ~switch[7];
   wire    my_we = sw_ntsc ? (hcount[0]==1'd1) : blank;
   wire [18:0]   write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]   write_data = sw_ntsc ? ntsc_data : vpat;

   // generate pixel value from reading ZBT memory
   wire [17:0]   vr_pixel;
   wire [18:0]   vram_addr1;

   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
               vram_addr1,vram_read_data);

//   wire  write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign      vram_addr = write_enable ? write_addr : vram_addr1;
//   assign      vram_we = write_enable;
      // used in ZBT Ram Module
   assign  vram_addr = my_we ? write_addr : vram_addr1;
   assign  vram_we = my_we;
   assign  vram_write_data = write_data;

      // Convert 24 bit RGB data into 24 bit HSV data
      wire [7:0] H,S,V;
      rgb2hsv RGB2HSV (.clock(tv_in_line_clock1), .reset(reset),
                                          .r({vr_pixel[17:12],2'd0}),
.g({vr_pixel[11:6],2'd0}),
                                          .b({vr_pixel[5:0],2'd0}), .h(H),
.s(S), .v(V));

      // Delay VGA signals other than pixel to line up with pixel delays
      wire delay_b,delay_hs,delay_vs;
      wire [17:0] delay_pixel;
      wire [10:0] delay_hcount;
      wire [9:0] delay_vcount;
      delayN delayB(.clk(clk), .in(blank), .out(delay_b));
      delayN delayHs(.clk(clk), .in(hsync), .out(delay_hs));
      delayN delayVs(.clk(clk), .in(vsync), .out(delay_vs));
      delayNbus18 delayPix(.clk(clk), .in(vr_pixel), .out(delay_pixel));
      delayNbus11 delayHcount(.clk(clk), .in(hcount),
.out(delay_hcount));
      delayNbus10 delayVcount(.clk(clk), .in(vcount),
.out(delay_vcount));

      // Raise a flag if current pixel matches ball color range
      wire [17:0] hue_pixel;
      wire match;
      wire [1:0] color;
      wire background;
      wire [12:0] x_center;
      wire [11:0] y_center;
      reg [10:0] x_center_test;
      reg [9:0] y_center_test;
      reg [31:0] middleHSV; // For display bank

      always @(posedge clk) begin
            // For narrowing the window we look at for COM calculation
            x_center_test <= button3_clean ? 0 : x_center;
            y_center_test <= button3_clean ? 0 : y_center;
            if (delay_hcount==500 && delay_vcount==500)
```

```
                middleHSV <= {H, 4'b0, S, 4'b0, V};
      end

      assign color = {switch[1], switch[0]};
      assign background = switch[2];
      hue_detector color_tester(.clk(clk), .H(H), .S(S), .V(V),
.color(color), .background(background),
                                                .hcount(delay_hcount),
.vcount(delay_vcount),.x_center(x_center_test),

      .y_center(y_center_test),.match(match), .rgb(delay_pixel),
.test_rgb(hue_pixel));

      // Calculate the Center of mass of the frame
      wire [23:0] matches_in_frame;
      wire center_done;
      wire average;
      assign average = 1;
      center_of_mass center(.average(average),.clk(clk), .match(match),
.hcount(delay_hcount), .vcount(delay_vcount),
                                            .x_center(x_center),
.y_center(y_center),

      .final_count(matches_in_frame), .done(center_done)
                                            );

      wire [9:0] threshold_height;
      wire [9:0] table_height;
      wire [2:0] state;
      wire record;
      wire point;
      wire [3:0] player_1_score;
      wire [3:0] player_2_score;

      FSM fsm(.clk(clk), .reset(reset), .up(button_up_clean),
.down(button_down_clean),
                        .throw_start(button1_clean),
.point_enter(button0_clean), .point_toggle(switch[4]),
                        .replay_start(switch[5]), .line_toggle(switch[6]),
.y_center(y_center),
                        .threshold_height(threshold_height),
.table_height(table_height),
                        .state(state), .record(record), .point(point),
                        .player_1_score(player_1_score),
.player_2_score(player_2_score));

      reg [8:0] addr_bram;
      reg clear_addr_bram;
      reg speed;
      reg center_done_2;
      reg speed_toggle;
      always @(posedge center_done) begin
            speed <= speed + 1;
            if (speed) center_done_2 <= 1;
            else if (speed == 0) center_done_2 <= 0;

            if (button_right_clean) speed_toggle <= ~speed_toggle;

            if ((record)&(~clear_addr_bram)) begin
                  addr_bram <= 0;
```

```
                clear_addr_bram <= 1;
            end
            else if ((record)&(clear_addr_bram)) begin
                addr_bram <= addr_bram + 1;
            end
            else if ((~record)&(clear_addr_bram)) begin
                addr_bram <= 0;
                clear_addr_bram <=0;
            end
            else if ((~record)&(switch[5])) begin
                if (speed_toggle)
                    addr_bram <= speed ? addr_bram + 1 : addr_bram;
                else
                    addr_bram <= addr_bram + 1;
            end
        end
        wire [24:0] mem_out;

// example use: make a 64K x 8 memory, this instance is 2^9 x 25 memory
        wire center_done_speed;
        wire center_done_slow;
        assign center_done_slow = speed_toggle ? center_done_2 :
center_done;
        assign center_done_speed = switch[5] ? center_done_slow :
center_done;
        mybram #(.LOGSIZE(9),.WIDTH(25))
          example(.addr(addr_bram),.clk(center_done_speed),

        .we(record),.din({x_center,y_center}),.dout(mem_out));
        wire [12:0] x_center_disp;
        wire [11:0] y_center_disp;
        assign x_center_disp = switch[5] ? mem_out[24:12] : x_center;
        assign y_center_disp = switch[5] ? mem_out[11:0] : y_center;


        // Generate sprite based on calcualted center of mass of hue
        wire [17:0] ball_pixel;
        ball_generator ball(.x_center(x_center_disp),
.y_center(y_center_disp),

        .hcount(delay_hcount),.vcount(delay_vcount),
                                                .matches(matches_in_frame),
.replay(switch[5]),
                                                .background(hue_pixel),
.pixel(ball_pixel));

        // Generate a Low line on screen
        wire [17:0] low_pixel;
        line_generator lowLine(.height(threshold_height), .color(1'b1),

        .hcount(delay_hcount),.vcount(delay_vcount),
                                                .background(ball_pixel),
.pixel(low_pixel));
        // Generate a Table line on screen
        wire [17:0] table_pixel;
        line_generator tableLine(.height(table_height), .color(1'b0),

        .hcount(delay_hcount),.vcount(delay_vcount),
                                                .background(low_pixel),
.pixel(table_pixel));
```

```
      // Generate a indicator of valid throw height
      wire [17:0] marker_pixel;
      low_indicator lowMarker(.point(point),
                                                .x_start(11'd30),
.y_start(10'd30),

      .hcount(delay_hcount),.vcount(delay_vcount),
                                                .background(table_pixel),
.pixel(marker_pixel));

      // Generate a Score on screen
      wire [17:0] score_pixel;
      scoreboard score(.clk(clk), .left_score(player_2_score),
.right_score(player_1_score),

      .hcount(delay_hcount),.vcount(delay_vcount),
                                          .background(marker_pixel),
.pixel(score_pixel));

      // Write the Game Over text on the screen when FSM in game over
state
      reg game_over;
      wire left_wins; // left will be red, right will be blue
      assign left_wins = (player_2_score > player_1_score);
      always @(posedge clk)begin
            if (state == 7)
                  game_over <= 1;
            else
                  game_over <= 0;
      end
      wire [17:0] word_pixel;
      Word #(.XSTART(20), .YSTART(400), .THICKNESS(20))
            gameOver(.hcount(delay_hcount),.vcount(delay_vcount),
                        .background(score_pixel), .pixel(word_pixel),
                        .red_wins(left_wins));

   // select output pixel data
   reg [17:0]    pixel;
   reg     b,hs,vs;

   always @(posedge clk)begin
      // Set output pixel to either black and white bars, or pixel read
from VRAM
      //    pixel <= switch[0] ?
{hcount[8:6],3'b0,hcount[8:6],3'b0,hcount[8:6],3'b0} : vr_pixel;
            pixel <= game_over ? word_pixel : score_pixel;
            b <= delay_b;
            hs <= delay_hs;
            vs <= delay_vs;
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clk.
   assign vga_out_red = {pixel[17:12],2'b0};
   assign vga_out_green = {pixel[11:6],2'b0};
   assign vga_out_blue = {pixel[5:0],2'b0};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clk;
   assign vga_out_blank_b = ~b;
```

```
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   // debugging

//   assign led = ~{vram_addr[18:13],reset,switch[0]};
//   assign led = ~{state,2'b0,ready,sound_on,reset};
      assign led = ~{state, 2'b0, button3_clean, sound_on, reset};
      reg [31:0] dispcount = 0;
      always @(posedge clk)begin
            if (dispcount == 27000000)begin
                  dispcount <= 0;
//                dispdata <= {low_audio, 2'b0, sound_addr, middleHSV,
4'b0, 1'b0, state};
                  dispdata <= {middleHSV ,29'b0, state};
            end
            else
                  dispcount <= dispcount + 1;
      end
endmodule

////////////////////////////////////////////////////////////////////////
//////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output  vsync;
   output  hsync;
   output  blank;

   reg       hsync,vsync,hblank,vblank,blank;
   reg [10:0]     hcount;    // pixel number on current line
   reg [9:0] vcount;    // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
```

```
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

/*
//////////////////////////////////////////////////////////////////////
//////
// xvga: Generate XVGA display signals (800 x 600 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output  vsync;
   output  hsync;
   output  blank;

   reg       hsync,vsync,hblank,vblank,blank;
   reg [10:0]     hcount;    // pixel number on current line
   reg [9:0] vcount;    // line number

   // horizontal: 1056 pixels total
   // display 800 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 799);
   assign    hsyncon = (hcount == 839);
   assign    hsyncoff = (hcount == 967);
   assign    hreset = (hcount == 1055);

   // vertical: 628 lines total
   // display 600 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 599);
   assign    vsyncon = hreset & (vcount == 600);
   assign    vsyncoff = hreset & (vcount == 604);
   assign    vreset = hreset & (vcount == 627);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule */
```

```
//////////////////////////////////////////////////////////////////////
////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.
//
// Bug due to memory management will be fixed. The bug happens because
// memory is called based on current hcount & vcount, which will actually
// shows up 2 cycle in the future. Not to mention that these incoming
data
// are latched for 2 cycles before they are used. Also remember that the
// ntsc2zbt's addressing protocol has been fixed.

// The original bug:
// -. At (hcount, vcount) = (100, 201) data at memory address(0,100,49)
//    arrives at vram_read_data, latch it to vr_data_latched.
// -. At (hcount, vcount) = (100, 203) data at memory address(0,100,49)
//    is latched to last_vr_data to be used for display.
// -. Remember that memory address(0,100,49) contains camera data
//    pixel(100,192) - pixel(100,195).
// -. At (hcount, vcount) = (100, 204) camera pixel data(100,192) is
shown.
// -. At (hcount, vcount) = (100, 205) camera pixel data(100,193) is
shown.
// -. At (hcount, vcount) = (100, 206) camera pixel data(100,194) is
shown.
// -. At (hcount, vcount) = (100, 207) camera pixel data(100,195) is
shown.
//
// Unfortunately this means that at (hcount == 0) to (hcount == 11) data
from
// the right side of the camera is shown instead (including possible sync
signals).

// To fix this, two corrections has been made:
// -. Fix addressing protocol in ntsc_to_zbt module.
// -. Forecast hcount & vcount 8 clock cycles ahead and use that
//    instead to call data from ZBT.


module vram_display(reset,clk,hcount,vcount,vr_pixel,
                vram_addr,vram_read_data);

   input reset, clk;
   input [10:0] hcount;
   input [9:0]   vcount;
   output [17:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   //forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
   wire [10:0] hcount_f = (hcount >= 1048) ? (hcount - 1048) : (hcount +
8);
   wire [9:0] vcount_f = (hcount >= 1048) ? ((vcount == 805) ? 0 : vcount
+ 1) : vcount;
```

```
   wire [18:0]    vram_addr = {vcount_f, hcount_f[9:1]};

   wire hc4 = hcount[0];
   reg [17:0]     vr_pixel;
   reg [35:0]     vr_data_latched;
   reg [35:0]     last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc4==1'd1) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4==1'd0) ? vram_read_data : vr_data_latched;

   always @(*)         // each 36-bit word from RAM is decoded to 4 bytes
     case (hc4)
       //2'd3: vr_pixel = last_vr_data[7:0];
       //2'd2: vr_pixel = last_vr_data[7+8:0+8];
       2'd1: vr_pixel = last_vr_data[17:0];
       2'd0: vr_pixel = last_vr_data[17+18:0+18];
     endcase

endmodule // vram_display

//////////////////////////////////////////////////////////////////////
////
// parameterized delay line

//module delayN(clk,in,out);
//   input clk;
//   input in;
//   output out;
//
//   parameter NDELAY = 3;
//
//   reg [NDELAY-1:0] shiftreg;
//   wire      out = shiftreg[NDELAY-1];
//
//   always @(posedge clk)
//     shiftreg <= {shiftreg[NDELAY-2:0],in};
//
//endmodule // delayN

// delay 1 bit by N clock cycles, used by b, hs, vs
module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 56;

   reg [NDELAY-1:0] shiftreg;
   wire        out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN

// delay 18 bits by N clock cycles
module delayNbus18(clk,in,out);
```

```
   input clk;
   input [17:0] in;
   output [17:0] out;

   parameter NDELAY = 56;
      parameter BUS = 18;

   reg [NDELAY*BUS-1:0] shiftreg;
   wire        out = shiftreg[BUS*NDELAY-1:BUS*(NDELAY-1)];

   always @(posedge clk)
     shiftreg <= {shiftreg[BUS*(NDELAY-1)-1:0],in};

endmodule // delayN

// delay 11 bits by N clock cycles
module delayNbus11(clk,in,out);
   input clk;
   input [10:0] in;
   output [10:0] out;

   parameter NDELAY = 56;
      parameter BUS = 11;

   reg [NDELAY*BUS-1:0] shiftreg;
   wire        out = shiftreg[BUS*NDELAY-1:BUS*(NDELAY-1)];

   always @(posedge clk)
     shiftreg <= {shiftreg[BUS*(NDELAY-1)-1:0],in};

endmodule // delayN

// delay 10 bits by N clock cycles
module delayNbus10(clk,in,out);
   input clk;
   input [9:0] in;
   output [9:0] out;

   parameter NDELAY = 56;
      parameter BUS = 10;

   reg [NDELAY*BUS-1:0] shiftreg;
   wire        out = shiftreg[BUS*NDELAY-1:BUS*(NDELAY-1)];

   always @(posedge clk)
     shiftreg <= {shiftreg[BUS*(NDELAY-1)-1:0],in};

endmodule // delayN


//////////////////////////////////////////////////////////////////////////////
///
// ramclock module

//////////////////////////////////////////////////////////////////////////////
//////
//
// 6.111 FPGA Labkit -- ZBT RAM clock generation
//
//
```

```
// Created: April 27, 2004
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////
//////
//
// This module generates deskewed clocks for driving the ZBT SRAMs and
FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so
that
// rising clock edges reach the RAMs at exactly the same time as rising
clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as
it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-
level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after
configuration.
// This is done using a 16-bit shift register. When the DCMs have locked,
the
// <lock> output of this mnodule will go high. Until the DCMs are locked,
the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//
//////////////////////////////////////////////////////////////////////////
//////

module ramclock(ref_clock, fpga_clock, ram0_clock, ram1_clock,
              clock_feedback_in, clock_feedback_out, locked);

   input ref_clock;                 // Reference clock input
   output fpga_clock;               // Output clock to drive FPGA logic
   output ram0_clock, ram1_clock;   // Output clocks for each RAM chip
   input  clock_feedback_in;        // Output to feedback trace
   output clock_feedback_out;       // Input from feedback trace
   output locked;                   // Indicates that clock outputs are
stable

   wire  ref_clk, fpga_clk, ram_clk, fb_clk, lock1, lock2, dcm_reset;


//////////////////////////////////////////////////////////////////////////
///

   //To force ISE to compile the ramclock, this line has to be removed.
   //IBUFG ref_buf (.O(ref_clk), .I(ref_clock));

      assign ref_clk = ref_clock;

   BUFG int_buf (.O(fpga_clock), .I(fpga_clk));

   DCM int_dcm (.CLKFB(fpga_clock),
```

```
            .CLKIN(ref_clk),
            .RST(dcm_reset),
            .CLK0(fpga_clk),
            .LOCKED(lock1));
   // synthesis attribute DLL_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of int_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of int_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of int_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of int_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of int_dcm is 0

   BUFG ext_buf (.O(ram_clock), .I(ram_clk));

   IBUFG fb_buf (.O(fb_clk), .I(clock_feedback_in));

   DCM ext_dcm (.CLKFB(fb_clk),
                .CLKIN(ref_clk),
                .RST(dcm_reset),
                .CLK0(ram_clk),
                .LOCKED(lock2));
   // synthesis attribute DLL_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of ext_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of ext_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of ext_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of ext_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of ext_dcm is 0

   SRL16 dcm_rst_sr (.D(1'b0), .CLK(ref_clk), .Q(dcm_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   // synthesis attribute init of dcm_rst_sr is "000F";


   OFDDRRSE ddr_reg0 (.Q(ram0_clock), .C0(ram_clock), .C1(~ram_clock),
                  .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
   OFDDRRSE ddr_reg1 (.Q(ram1_clock), .C0(ram_clock), .C1(~ram_clock),
                  .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
   OFDDRRSE ddr_reg2 (.Q(clock_feedback_out), .C0(ram_clock),
.C1(~ram_clock),
                  .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));

   assign locked = lock1 && lock2;

endmodule




\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
module YCrCb2RGB ( R, G, B, clk, rst, Y, Cr, Cb );

output [7:0]  R, G, B;

input clk,rst;
input[9:0] Y, Cr, Cb;

wire [7:0] R,G,B;
reg [20:0] R_int,G_int,B_int,X_int,A_int,B1_int,B2_int,C_int;
reg [9:0] const1,const2,const3,const4,const5;
```

```
reg[9:0] Y_reg, Cr_reg, Cb_reg;

//registering constants
always @ (posedge clk)
begin
 const1 = 10'b 0100101010; //1.164 = 01.00101010
 const2 = 10'b 0110011000; //1.596 = 01.10011000
 const3 = 10'b 0011010000; //0.813 = 00.11010000
 const4 = 10'b 0001100100; //0.392 = 00.01100100
 const5 = 10'b 1000000100; //2.017 = 10.00000100
end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
   else
      begin
        Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     A_int <= (const2 * (Cr_reg - 'd512));
     B1_int <= (const3 * (Cr_reg - 'd512));
     B2_int <= (const4 * (Cb_reg - 'd512));
     C_int <= (const5 * (Cb_reg - 'd512));
     end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     R_int <= X_int + A_int;
     G_int <= X_int - B1_int - B2_int;
     B_int <= X_int + C_int;
     end



/*always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     R_int <= X_int + (const2 * (Cr_reg - 'd512));
```

```
     G_int <= X_int - (const3 * (Cr_reg - 'd512)) - (const4 * (Cb_reg -
'd512));
     B_int <= X_int + (const5 * (Cb_reg - 'd512));
     end

*/
/* limit output to 0 - 4095, <0 equals o and >4095 equals 4095 */
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] :
8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] :
8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] :
8'b11111111;

endmodule




////////////////////////////////////////////////////////////////////////
/////////
module Word # (parameter XSTART=0, YSTART=0, THICKNESS=10,
                          RED={6'h3f,6'h00,6'h00},
                          GREEN={6'h00,6'h3f,6'h00},
                          BLUE={6'h00,6'h00,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output [17:0] pixel,
       input red_wins
    );

    wire [17:0] b_pixel, l_pixel, u_pixel, e_pixel,
                                  w_pixel,o_pixel,n_pixel, r_pixel,
e2_pixel, d_pixel;

      // WON
    W # (.XSTART(XSTART+550), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(GREEN))
      w(.hcount(hcount), .vcount(vcount), .background(background),
.pixel(w_pixel));

    O # (.XSTART(XSTART+725), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(GREEN))
      o(.hcount(hcount), .vcount(vcount), .background(w_pixel),
.pixel(o_pixel));

    N # (.XSTART(XSTART+860), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(GREEN))
      n(.hcount(hcount), .vcount(vcount), .background(o_pixel),
.pixel(n_pixel));

      // BLUE
    B # (.XSTART(XSTART), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(BLUE))
      b(.hcount(hcount), .vcount(vcount), .background(n_pixel),
.pixel(b_pixel));
```

```
    L # (.XSTART(XSTART+125), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(BLUE))
      l(.hcount(hcount), .vcount(vcount), .background(b_pixel),
.pixel(l_pixel));

    U # (.XSTART(XSTART+250), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(BLUE))
      u(.hcount(hcount), .vcount(vcount), .background(l_pixel),
.pixel(u_pixel));

    E # (.XSTART(XSTART+375), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(BLUE))
      e(.hcount(hcount), .vcount(vcount), .background(u_pixel),
.pixel(e_pixel));

      // RED
    R # (.XSTART(XSTART+50), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(RED))
      r(.hcount(hcount), .vcount(vcount), .background(n_pixel),
.pixel(r_pixel));

    E # (.XSTART(XSTART+175), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(RED))
      e2(.hcount(hcount), .vcount(vcount), .background(r_pixel),
.pixel(e2_pixel));

    D # (.XSTART(XSTART+300), .YSTART(YSTART), .THICKNESS(THICKNESS),
.COLOR(RED))
      d(.hcount(hcount), .vcount(vcount), .background(e2_pixel),
.pixel(d_pixel));

    assign pixel = red_wins ? d_pixel : e_pixel;

endmodule

// B
module B # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
        // Letter will be contained in 75 x 150 block within 125 x 200
block
            if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount
> YSTART + 25 && vcount < YSTART + 175)) begin
             if (hcount > XSTART + 75)begin
                 if (vcount < YSTART + 100)begin
                     if ((hcount + YSTART > vcount + XSTART + 50) ||
(hcount + vcount > 175 + XSTART + YSTART))
                         pixel = background;
                     else
                         pixel = COLOR;
                 end
                 else begin
```

```
                    if ((hcount + YSTART + 25 > vcount + XSTART) ||
(hcount + vcount > 250 + XSTART + YSTART))
                        pixel = background;
                    else
                        pixel = COLOR;
                end
            end
            else if ((hcount < XSTART + 50))
                pixel = COLOR;
            else if ((vcount < YSTART + 50) || (vcount > YSTART + 150))
                pixel = COLOR;
                      else if ((vcount > YSTART + 90) && (vcount <
YSTART + 110))
                            pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule


// D
module D # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
        // Letter will be contained in 75 x 150 block within 125 x 200
block
     if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if (hcount > XSTART + 75)begin
                if ((hcount + YSTART > vcount + XSTART + 50) || (hcount +
vcount > 250 + XSTART + YSTART))
                    pixel = background;
                else
                    pixel = COLOR;
            end
            else if ((hcount < XSTART + 50))
                pixel = COLOR;
            else if ((vcount < YSTART + 50) || (vcount > YSTART + 150))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule
```

```
// E
module E # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
        // Letter N will be contained in 75 x 150 block within 125 x 200
block
      if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount < XSTART + 50))
                pixel = COLOR;
            else if ((vcount < YSTART + 50) || (vcount > YSTART + 150) ||
((vcount > YSTART + 90) && (vcount < YSTART + 110)))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule

// L
module L # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
        // Letter will be contained in 75 x 150 block within 125 x 200
block
      if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount < XSTART + 50))
                pixel = COLOR;
            else if ((vcount > YSTART + 150))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule
```

```
// N
module N # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    wire [11:0] double_hcount;
    assign double_hcount = (hcount-XSTART) << 1;
    always @ * begin
        // Letter will be contained in 75 x 150 block within 125 x 200
block
      if ((hcount > XSTART + 15 && hcount < XSTART + 110) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount < XSTART + 40) || (hcount > XSTART + 85))
                pixel = COLOR;
            else if(((double_hcount) + YSTART + THICKNESS > vcount + 25)
&& ((double_hcount) + YSTART < THICKNESS + vcount + 25))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule


// O
module O # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
        // Letter N will be contained in 75 x 150 block within 125 x 200
block
      if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount < XSTART + 50) || (hcount > XSTART + 75))
                pixel = COLOR;
            else if ((vcount < YSTART + 50) || (vcount > YSTART + 150))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end
```

```
endmodule

//R
module R # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    wire [11:0] double_hcount;
    assign double_hcount = (hcount-XSTART) << 1;
    always @ * begin
        // Letter will be contained in 75 x 150 block within 125 x 200
block
      if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount > XSTART + 75) && (vcount < YSTART + 110)) begin
                if ((hcount + YSTART > vcount + XSTART + 50) || (hcount +
vcount > 185 + XSTART + YSTART))
                    pixel = background;
                else
                    pixel = COLOR;
            end
            else if ((hcount < XSTART + 50))
                pixel = COLOR;
                       else if ((vcount < YSTART + 50) || ((vcount >
YSTART + 90) && (vcount < YSTART + 110)))
                                pixel = COLOR;
            else if(((double_hcount) + YSTART + THICKNESS > vcount + 25)
&&
                                            ((double_hcount) + YSTART <
THICKNESS + vcount + 25) &&
                                            (vcount > YSTART + 100))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule

// U
module U # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
```

```
        // Letter will be contained in 75 x 150 block within 125 x 200
block
      if ((hcount > XSTART + 25 && hcount < XSTART + 100) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount < XSTART + 50) || (hcount > XSTART + 75))
                pixel = COLOR;
            else if ((vcount > YSTART + 150))
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule

//W
module W # (parameter XSTART=0, YSTART=0, THICKNESS=10,
COLOR={6'h3f,6'h3f,6'h3f})
      (
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

    always @ * begin
        // Letter W will be contained in 125 x 150 block within 175 x 200
block
      if ((hcount > XSTART + 25 && hcount < XSTART + 150) && (vcount >
YSTART + 25 && vcount < YSTART + 175)) begin
            if ((hcount < XSTART + 50) || (hcount > XSTART + 125))
                pixel = COLOR;
            else if ((hcount > XSTART + 75) && (hcount < XSTART + 100))
                pixel = COLOR;
            else if (vcount > YSTART + 150)
                pixel = COLOR;
            else
                pixel = background;
        end
        else begin
            pixel = background;
        end
    end

endmodule


//////////////////////////////////////////////////////////////////////////
/////////
module scoreboard #(parameter RED={6'h3f,6'h00,6'h00},
BLUE={6'h00,6'h00,6'h3f})
      (
    input clk,
    input [3:0] left_score,
    input [3:0] right_score,
    input [10:0] hcount,
    input [9:0] vcount,
```

```
    input [17:0] background,
    output [17:0] pixel
    );

      wire [17:0] left_pixel, right_pixel, dash_pixel;
      score_generator # (.XSTART(300), .YSTART(30), .COLOR(RED))
            leftScore(.clk(clk), .score(left_score),
                                 .hcount(hcount),.vcount(vcount),
                                 .background(background),
.pixel(left_pixel));
      score_generator # (.XSTART(600), .YSTART(30), .COLOR(BLUE))
            rightScore(.clk(clk), .score(right_score),
                                 .hcount(hcount),.vcount(vcount),
                                 .background(left_pixel),
.pixel(right_pixel));
      score_generator # (.XSTART(450), .YSTART(30))
            dash(.clk(clk), .score(4'hF),
                        .hcount(hcount),.vcount(vcount),
                        .background(right_pixel), .pixel(dash_pixel));

      assign pixel = dash_pixel;

endmodule

//////////////////////////////////////////////////////////////////////////////
/////////
module score_generator #(parameter XSTART=0, YSTART=0,
COLOR={6'h3f,6'h3f,6'h3f})
      (
       input clk,
       input [3:0] score, // 0-9 correspond to 0-9, otherwise dash
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
    output reg [17:0] pixel
    );

      reg A,B,C,D,E,F,G;
      // Designate the segements corresponding to the number we want to
display
      always @(posedge clk)begin
            case(score)
                  4'h0:begin
                        A <= 1;
                        B <= 1;
                        C <= 1;
                        D <= 1;
                        E <= 1;
                        F <= 1;
                        G <= 0;
                  end
                  4'h1:begin
                        A <= 0;
                        B <= 1;
                        C <= 1;
                        D <= 0;
                        E <= 0;
                        F <= 0;
                        G <= 0;
                  end
```

```
4'h2:begin
      A <= 1;
      B <= 1;
      C <= 0;
      D <= 1;
      E <= 1;
      F <= 0;
      G <= 1;
end
4'h3:begin
      A <= 1;
      B <= 1;
      C <= 1;
      D <= 1;
      E <= 0;
      F <= 0;
      G <= 1;
end
4'h4:begin
      A <= 0;
      B <= 1;
      C <= 1;
      D <= 0;
      E <= 0;
      F <= 1;
      G <= 1;
end
4'h5:begin
      A <= 1;
      B <= 0;
      C <= 1;
      D <= 1;
      E <= 0;
      F <= 1;
      G <= 1;
end
4'h6:begin
      A <= 1;
      B <= 0;
      C <= 1;
      D <= 1;
      E <= 1;
      F <= 1;
      G <= 1;
end
4'h7:begin
      A <= 1;
      B <= 1;
      C <= 1;
      D <= 0;
      E <= 0;
      F <= 0;
      G <= 0;
end
4'h8:begin
      A <= 1;
      B <= 1;
      C <= 1;
      D <= 1;
      E <= 1;
```

```
                        F <= 1;
                        G <= 1;
                end
                4'h9:begin
                        A <= 1;
                        B <= 1;
                        C <= 1;
                        D <= 1;
                        E <= 0;
                        F <= 1;
                        G <= 1;
                end
                default:begin
                        A <= 0;
                        B <= 0;
                        C <= 0;
                        D <= 0;
                        E <= 0;
                        F <= 0;
                        G <= 1;
                end
        endcase
end
// Fill in the appropriate segments
always @* begin
        if ((hcount > XSTART + 20 && hcount < XSTART + 80) &&
        (vcount > YSTART + 0 && vcount < YSTART + 20))
                pixel = A ? COLOR : background;
        else if ((hcount > XSTART + 80 && hcount < XSTART + 100) &&
        (vcount > YSTART + 0 && vcount < YSTART + 100))
                pixel = B ? COLOR : background;
        else if ((hcount > XSTART + 80 && hcount < XSTART + 100) &&
        (vcount > YSTART + 100 && vcount < YSTART + 200))
                pixel = C ? COLOR : background;
        else if ((hcount > XSTART + 20 && hcount < XSTART + 80) &&
        (vcount > YSTART + 180 && vcount < YSTART + 200))
                pixel = D ? COLOR : background;
        else if ((hcount > XSTART + 0 && hcount < XSTART + 20) &&
        (vcount > YSTART + 100 && vcount < YSTART + 200))
                pixel = E ? COLOR : background;
        else if ((hcount > XSTART + 0 && hcount < XSTART + 20) &&
        (vcount > YSTART + 0 && vcount < YSTART + 100))
                pixel = F ? COLOR : background;
        else if ((hcount > XSTART + 20 && hcount < XSTART + 80) &&
        (vcount > YSTART + 90 && vcount < YSTART + 110))
                pixel = G ? COLOR : background;
        else
                pixel = background;
end

endmodule

//////////////////////////////////////////////////////////////////////////////////
/////////
module rgb2hsv(clock, reset, r, g, b, h, s, v);
        input wire clock;
        input wire reset;
        input wire [7:0] r;
        input wire [7:0] g;
        input wire [7:0] b;
```

```
            output reg [7:0] h;
            output reg [7:0] s;
            output reg [7:0] v;
            reg [7:0] my_r_delay1, my_g_delay1, my_b_delay1;
            reg [7:0] my_r_delay2, my_g_delay2, my_b_delay2;
            reg [7:0] my_r, my_g, my_b;
            reg [7:0] min, max, delta;
            reg [15:0] s_top;
            reg [15:0] s_bottom;
            reg [15:0] h_top;
            reg [15:0] h_bottom;
            wire [15:0] s_quotient;
            wire [15:0] s_remainder;
            wire s_rfd;
            wire [15:0] h_quotient;
            wire [15:0] h_remainder;
            wire h_rfd;
            reg [7:0] v_delay [19:0];
            reg [18:0] h_negative;
            reg [15:0] h_add [18:0];
            reg [4:0] i;
            // Clocks 4-18: perform all the divisions
            //the s_divider (16/16) has delay 18
            //the hue_div (16/16) has delay 18

            coreGenDivider hue_div1(
            .clk(clock),
            .dividend(s_top),
            .divisor(s_bottom),
            .quotient(s_quotient),
              // note: the "fractional" output was originally named
"remainder" in this
            // file -- it seems coregen will name this output "fractional"
even if
            // you didn't select the remainder type as fractional.
            .fractional(s_remainder),
            .rfd(s_rfd)
            );
            coreGenDivider hue_div2(
            .clk(clock),
            .dividend(h_top),
            .divisor(h_bottom),
            .quotient(h_quotient),
            .fractional(h_remainder),
            .rfd(h_rfd)
            );
            always @ (posedge clock) begin

                  // Clock 1: latch the inputs (always positive)
                  {my_r, my_g, my_b} <= {r, g, b};

                  // Clock 2: compute min, max
                  {my_r_delay1, my_g_delay1, my_b_delay1} <= {my_r, my_g,
my_b};

                  if((my_r >= my_g) && (my_r >= my_b)) //(B,S,S)
                        max <= my_r;
                  else if((my_g >= my_r) && (my_g >= my_b)) //(S,B,S)
                        max <= my_g;
                  else  max <= my_b;
```

```
                if((my_r <= my_g) && (my_r <= my_b)) //(S,B,B)
                      min <= my_r;
                else if((my_g <= my_r) && (my_g <= my_b)) //(B,S,B)
                      min <= my_g;
                else
                      min <= my_b;

                // Clock 3: compute the delta
                {my_r_delay2, my_g_delay2, my_b_delay2} <= {my_r_delay1,
my_g_delay1, my_b_delay1};
                v_delay[0] <= max;
                delta <= max - min;

                // Clock 4: compute the top and bottom of whatever
divisions we need to do
                s_top <= 8'd255 * delta;
                s_bottom <= (v_delay[0]>0)?{8'd0, v_delay[0]}: 16'd1;


                if(my_r_delay2 == v_delay[0]) begin
                      h_top <= (my_g_delay2 >= my_b_delay2)?(my_g_delay2
- my_b_delay2) * 8'd255:(my_b_delay2 - my_g_delay2) * 8'd255;
                      h_negative[0] <= (my_g_delay2 >= my_b_delay2)?0:1;
                      h_add[0] <= 16'd0;
                end
                else if(my_g_delay2 == v_delay[0]) begin
                      h_top <= (my_b_delay2 >= my_r_delay2)?(my_b_delay2
- my_r_delay2) * 8'd255:(my_r_delay2 - my_b_delay2) * 8'd255;
                      h_negative[0] <= (my_b_delay2 >= my_r_delay2)?0:1;
                      h_add[0] <= 16'd85;
                end
                else if(my_b_delay2 == v_delay[0]) begin
                      h_top <= (my_r_delay2 >= my_g_delay2)?(my_r_delay2
- my_g_delay2) * 8'd255:(my_g_delay2 - my_r_delay2) * 8'd255;
                      h_negative[0] <= (my_r_delay2 >= my_g_delay2)?0:1;
                      h_add[0] <= 16'd170;
                end

                h_bottom <= (delta > 0)?delta * 8'd6:16'd6;


                //delay the v and h_negative signals 18 times
                for(i=1; i<19; i=i+1) begin
                      v_delay[i] <= v_delay[i-1];
                      h_negative[i] <= h_negative[i-1];
                      h_add[i] <= h_add[i-1];
                end

                v_delay[19] <= v_delay[18];
                //Clock 22: compute the final value of h
                //depending on the value of h_delay[18], we need to
subtract 255 from it to make it come back around the circle
                if(h_negative[18] && (h_quotient > h_add[18])) begin
                      h <= 8'd255 - h_quotient[7:0] + h_add[18];
                end
                else if(h_negative[18]) begin
                      h <= h_add[18] - h_quotient[7:0];
                end
                else begin
```

```
                    h <= h_quotient[7:0] + h_add[18];
                end

                //pass out s and v straight
                s <= s_quotient;
                v <= v_delay[19];
            end
endmodule
//////////////////////////////////////////////////////////////////////////
////
// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data,
ntsc_we, sw);

   input    clk; // system clock
   input    vclk;     // video clock from camera
   input [2:0]    fvh;
   input    dv;
   input [17:0]   din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output   ntsc_we;  // write enable for NTSC data
   input    sw;       // switch which determines mode (for debugging)


   //parameter    COL_START = 10'd150;
   //parameter    ROW_START = 10'd60;
   parameter      COL_START = 0;
   parameter      ROW_START = 0;

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 * 788 XGA display

   reg [9:0]      col = 0; // moved to outputs to use outside module
   reg [9:0]      row = 0;
   reg [17:0]     vdata = 0;
   reg            vwe;
   reg            old_dv;
   reg            old_frame;// frames are even / odd interlaced
   reg            even_odd; // decode interlaced frame to this wire

   wire     frame = fvh[2];
   wire     frame_edge = frame & ~old_frame;

   always @ (posedge vclk) //LLC1 is reference
     begin
      old_dv <= dv;
      vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
      old_frame <= frame;
      even_odd = frame_edge ? ~even_odd : even_odd;

      if (!fvh[2])
        begin
           col <= fvh[0] ? COL_START :
                (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 :
col;
           row <= fvh[1] ? ROW_START :
                (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
           vdata <= (dv && !fvh[2]) ? din : vdata;
```

```
      end
    end

  // synchronize with system clock

  reg [9:0] x[1:0],y[1:0];
  reg [17:0] data[1:0];
  reg       we[1:0];
  reg          eo[1:0];

  always @(posedge clk)
    begin
     {x[1],x[0]} <= {x[0],col};
     {y[1],y[0]} <= {y[0],row};
     {data[1],data[0]} <= {data[0],vdata};
     {we[1],we[0]} <= {we[0],vwe};
     {eo[1],eo[0]} <= {eo[0],even_odd};
    end

  // edge detection on write enable signal

  reg old_we;
  wire we_edge = we[1] & ~old_we;
  always @(posedge clk) old_we <= we[1];

  // shift each set of four bytes into a large register for the ZBT

  reg [35:0] mydata;
  always @(posedge clk)
    if (we_edge)
      mydata <= { mydata[17:0], data[1] };

  // NOTICE : Here we have put 4 pixel delay on mydata. For example,
when:
  // (x[1], y[1]) = (60, 80) and eo[1] = 0, then:
  // mydata[31:0] = ( pixel(56,160), pixel(57,160), pixel(58,160),
pixel(59,160) )
  // This is the root of the original addressing bug.


  // NOTICE : Notice that we have decided to store mydata, which
  //          contains pixel(56,160) to pixel(59,160) in address
  //          (0, 160 (10 bits), 60 >> 2 = 15 (8 bits)).
  //
  //          This protocol is dangerous, because it means
  //          pixel(0,0) to pixel(3,0) is NOT stored in address
  //          (0, 0 (10 bits), 0 (8 bits)) but is rather stored
  //          in address (0, 0 (10 bits), 4 >> 2 = 1 (8 bits)). This
  //          calculation ignores COL_START & ROW_START.
  //
  //          4 pixels from the right side of the camera input will
  //          be stored in address corresponding to x = 0.
  //
  //          To fix, delay col & row by 4 clock cycles.
  //          Delay other signals as well.

  reg [39:0] x_delay;
  reg [39:0] y_delay;
  reg [3:0] we_delay;
  reg [3:0] eo_delay;
```

```
   always @ (posedge clk)
   begin
     x_delay <= {x_delay[29:0], x[1]};
     y_delay <= {y_delay[29:0], y[1]};
     we_delay <= {we_delay[2:0], we[1]};
     eo_delay <= {eo_delay[2:0], eo[1]};
   end

//   // compute address to store data in
//   wire [8:0] y_addr = y_delay[38:30];
//    wire [9:0] x_addr = x_delay[39:30];
   // compute address to store data in
      parameter YOFFSET = 125;
      parameter XOFFSET = 150;
   wire [8:0] y_addr = (y_delay[38:30]+YOFFSET < 768) ?
(y_delay[38:30]+YOFFSET) : (y_delay[38:30]+YOFFSET - 768);
      wire [9:0] x_addr = (x_delay[39:30]+XOFFSET < 1024) ?
(x_delay[39:30]+XOFFSET) : (x_delay[39:30]+XOFFSET - 1024);

   //wire [18:0] myaddr = {1'b0,y_addr[8:0], eo_delay[1],x_addr[9:2]};
      wire [18:0] myaddr = {y_addr[8:0], eo_delay[1],x_addr[9:1]};

   // Now address (0,0,0) contains pixel data(0,0) etc.


   // alternate (256x192) image data and address
   wire [35:0] mydata2 = {data[1],data[1]};
   wire [18:0] myaddr2 = {y_addr[8:0], eo_delay[1],x_addr[8:0]};

   // update the output address and data only when four bytes ready

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==1'b0));

   always @(posedge clk)
     if ( ntsc_we )
                begin
        ntsc_addr <= sw ? myaddr2 : myaddr;  // normal and expanded modes
        ntsc_data <= sw ? mydata2 : mydata;
                end

endmodule // ntsc_to_zbt
////////////////////////////////////////////////////////////////////////
/////////
module mybram #(parameter LOGSIZE=14, WIDTH=1)
              (input wire [LOGSIZE-1:0] addr,
               input wire clk,
               input wire [WIDTH-1:0] din,
               output reg [WIDTH-1:0] dout,
               input wire we);
   // let the tools infer the right number of BRAMs
   (* ram_style = "block" *)
   reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
   always @(posedge clk) begin
           if (we) mem[addr] <= din;
           dout <= mem[addr];
   end
endmodule
```

```
////////////////////////////////////////////////////////////////////////////
/////////
module low_indicator(
      input point,
      input [10:0] x_start,
      input [9:0] y_start,
   input [10:0] hcount,
   input [9:0] vcount,
   input [17:0] background,
   output reg [17:0] pixel
    );

      parameter THICKNESS = 15;
      parameter GREEN = {6'h00,6'h3f,6'h00};
      parameter RED = {6'h3f,6'h00,6'h00};
      parameter XMAX = 200;
      parameter YMAX = 200;

      always @* begin
            if ((hcount > x_start && hcount < XMAX) &&
                  (vcount > y_start && vcount < YMAX)) begin
                  // Generate a check mark
                  if (point) begin
                        if ( ((hcount + 120 + THICKNESS > vcount) &&
(hcount + 120 < vcount + THICKNESS)) ||
                              ((hcount + vcount + THICKNESS > x_start +
240) && (hcount + vcount < x_start + 240 + THICKNESS)))
                              pixel = GREEN;
                        else
                              pixel = background;
                  end
                  // Generate an X
                  else begin
                        if ( ((hcount + THICKNESS > vcount) && (hcount <
vcount + THICKNESS)) ||
                              ((hcount + vcount + THICKNESS > x_start +
200) && (hcount + vcount < x_start + 200 + THICKNESS)))
                              pixel = RED;
                        else
                              pixel = background;
                  end
            end
            else
                  pixel = background;
      end

endmodule


////////////////////////////////////////////////////////////////////////////
/////////
module line_generator(
    input [9:0] height,
    input [10:0] hcount,
    input [9:0] vcount,
    input [17:0] background,
       input color,
    output reg [17:0] pixel
    );
       parameter THICKNESS = 5;
```

```
       parameter RED = {6'h3f,6'h00,6'h00};
       parameter GREEN = {6'h00,6'h3f,6'h00};
       parameter YOFFSET = 250;
       parameter XOFFSET = 150;
       parameter XMIN = 0 + XOFFSET;
       parameter XMAX = 720 + XOFFSET;

       always @* begin
      if ((hcount > XMIN && hcount < XMAX) &&
                  ((vcount+THICKNESS)>(height+YOFFSET) &&
vcount<(height+YOFFSET+THICKNESS)))
                  pixel = color ? RED : GREEN;
            else
                  pixel = background;
       end


endmodule

//////////////////////////////////////////////////////////////////////////////
/////////
module hue_detector(
    input clk,
    input [7:0] H,
       input [7:0] S,
       input [7:0] V,
    input [1:0] color,// expand as necessary for testing
       input background,
       input [17:0] rgb,
       input [10:0] hcount,
       input [9:0] vcount,
       input [10:0] x_center,
       input [9:0] y_center,
       output reg [17:0] test_rgb,
       output reg match
    );

       parameter XOFFSET = 150;
       parameter YOFFSET = 250;
       parameter XMIN = 0 + XOFFSET;
       parameter XMAX = 700 + XOFFSET;
       parameter YMIN = 10 + YOFFSET;
       parameter YMAX = 500 + YOFFSET;

      reg [10:0] X_Start = XMIN;
      reg [10:0] X_End = XMAX;
      reg [9:0] Y_Start = YMIN;
      reg [9:0] Y_End = YMAX;

      parameter MinS = 8'h30;
      parameter MaxS = 8'hB0;
      parameter MinV = 8'h30;
      parameter MaxV = 8'hF0;

      reg testerH = 0;
      reg testerS = 0;
      reg testerV = 0;

      always @(posedge clk)begin
            if ((x_center==0) && (y_center==0))begin
```

```
                X_Start <= XMIN;
                X_End <= XMAX;
                Y_Start <= YMIN;
                Y_End <= YMAX;
          end
          else begin
                X_Start <= (x_center<(XMIN+50)) ? XMIN : x_center - 50;
                X_End <= (x_center>(XMAX-50)) ? XMAX : x_center + 50;
                Y_Start <= (y_center<(YMIN+50)) ? YMIN : y_center - 50;
                Y_End <= (y_center>(YMAX-50)) ? YMAX : y_center + 50;
          end
          if((hcount>X_Start) && (hcount<X_End)
                && (vcount>Y_Start) && (vcount<Y_End))begin
                case(color)
                      // RED 00h < H < 20h
                      2'b00:begin
                            if(H > 8'h00 && H < 8'h18)begin
                                  testerH <= 1;
                                  if(S > MinS && S < MaxS)begin
                                        testerS <= 1;
                                        if(V > MinV && V < MaxV)
                                              testerV <= 1;
                                        else
                                              testerV <= 0;
                                  end
                                  else
                                        testerS <= 0;
                            end
                            else begin
                                  testerH <= 0;
                            end
                      end
                      // BLUE 96h < H < BEh
                      2'b01:begin
                            if(H > 8'h96 && H < 8'hBE)begin
                                  testerH <= 1;
                                  if(S > MinS && S < MaxS)begin
                                        testerS <= 1;
//                                      if(V > MinV && V < MaxV) // blue
ball had issues matching value
                                              testerV <= 1;
//                                    else
//                                          testerV <= 0;
                                  end
                                  else
                                        testerS <= 0;
                            end
                            else begin
                                  testerH <= 0;
                            end
                      end
                      // GREEN 20h < H < 3Fh
                      2'b10:begin
                            if(H > 8'h20 && H < 8'h3F)begin
                                  testerH <= 1;
                                  if(S > MinS && S < MaxS)begin
                                        testerS <= 1;
                                        if(V > MinV && V < MaxV)
                                              testerV <= 1;
                                        else
```

```
                                        testerV <= 0;
                                  end
                                  else
                                        testerS <= 0;
                            end
                            else begin
                                  testerH <= 0;
                            end
                      end
                      // YELLOW 10h < H < 2Fh
                      2'b11:begin
                            if(H > 8'h10 && H < 8'h2F)begin
                                  testerH <= 1;
                                  if(S > MinS && S < MaxS)begin
                                        testerS <= 1;
                                        if(V > MinV && V < MaxV)
                                              testerV <= 1;
                                        else
                                              testerV <= 0;
                                  end
                                  else
                                        testerS <= 0;
                            end
                            else begin
                                  testerH <= 0;
                            end
                      end
                      default:begin
                            testerH <= 0;
                      end
                endcase
          end
          // hcount and vcount outside of bounds
          else begin
                testerH <= 0;
          end

          // Now assign pixel values and set a match flag based on HSV
matching
          if (testerH)begin
                if (testerS)begin
                      if (testerV)begin
                            match <= 1;
                            test_rgb <= background ? {6'h3f,6'h00,6'h3f}
: rgb; // Magenta
                      end
                      else begin
                            match <= 0;
                            test_rgb <= background ? {6'h00,6'h00,6'h3f}
: rgb; // Blue
                      end
                end
                else begin
                      match <= 0;
                      test_rgb <= background ? {6'h00,6'h3f,6'h00} :
rgb; // Green
                end
          end
          else begin
                match <= 0;
```

```
                  test_rgb <= background ? {18'b0} : rgb;
            end
      end // end of always block

endmodule

//////////////////////////////////////////////////////////////////////////////
/////////
module FSM( input clk,
      input reset,
      input up,
      input down,
      input throw_start,
      input point_enter,
      input point_toggle,
      input replay_start,
      input line_toggle,
      input [10:0] y_center,

      output reg [9:0] threshold_height,
      output reg [9:0] table_height,
      output reg [2:0] state,
      output reg record,
      output reg point,
      output reg [3:0] player_1_score,
      output reg [3:0] player_2_score);

      reg [32:0] transition;
      reg [32:0] countdown;
      reg [64:0] countdown_2;
      reg debounce_1;
      reg debounce_2;

      parameter idle = 1;
      parameter replay = 2;
      parameter play_low_bf = 3;
      parameter play_high = 4;
      parameter play_low_af = 5;
      parameter record_end_play = 6;
      parameter game_over = 7;
      parameter buffer = 10;
      parameter point_to_end = 100000000;
      parameter time_out = 300000000;
      parameter YOFFSET = 250;

      initial begin
            state <= idle;
            transition <= 0;
            countdown <= 0;
            countdown_2 <= 0;
            threshold_height <= 75;
            table_height <= 400;
            player_1_score <= 0;
            player_2_score <= 0;
            debounce_1 <= 1;
            debounce_2 <= 1;
      end

      always @(posedge clk) begin
            if (reset) begin    //Reset all system variables
```

```
                state <= idle;
                transition <= 0;
                countdown <= 0;
                countdown_2 <= 0;
                threshold_height <= 75;
                table_height <= 400;
                player_1_score <= 0;
                player_2_score <= 0;
                record <= 0;
                point <= 0;
          end

          if (state == idle) begin               //user-inputs will now
be recognized
                countdown <= 0;
                countdown_2 <= 0;

                //Line toggling
                if ((line_toggle == 0)&(up)&(debounce_1)) begin
                      threshold_height <= threshold_height - 5;
                      debounce_1 <= 0;
                end
                else if ((line_toggle == 0)&(down)&(debounce_1)) begin
                      threshold_height <= threshold_height + 5;
                      debounce_1 <= 0;
                end
                else if ((line_toggle == 1)&(up)&(debounce_1)) begin
                      table_height <= table_height - 5;
                      debounce_1 <= 0;
                end
                else if ((line_toggle == 1)&(down)&(debounce_1)) begin
                      table_height <= table_height + 5;
                      debounce_1 <= 0;
                end
                else if ((~up)&(~down)) debounce_1 <= 1;


                //Point set-up
                if ((point_toggle == 0)&(point_enter)&(debounce_2))
begin
                      player_1_score <= player_1_score + 1;
                      debounce_2 <= 0;
                end
                else if ((point_toggle == 1)&(point_enter)&(debounce_2))
begin
                      player_2_score <= player_2_score + 1;
                      debounce_2 <= 0;
                end
                else if (~point_enter) debounce_2 <= 1;

                //Game-over Transition
                if (((player_1_score > player_2_score +
1)&(player_1_score > 6))|
                            ((player_2_score > player_1_score +
1)&(player_2_score > 6)))
                      state <= game_over;
     //endgame by 7, two points ahead

                //Replay Transition, or Start-Throw
```

```
                if (replay_start) state <= replay;
//begin replay
                else if (throw_start) begin                    //begin
throw and recording
                      state <= play_low_bf;
                      record <= 1;
                      point <= 0;
                end
          end

          if ((state == replay)&(replay_start == 0))
                state <= idle;  //transition back to idle after replay

          //Object still below threshold line
          if ((state == play_low_bf)&(y_center < threshold_height -
buffer + YOFFSET)) begin
                state <= play_high;
                point <= 1;
          end
          else if ((state == play_low_bf)&(y_center > table_height +
buffer + YOFFSET))
                state <= record_end_play;    //If it drops below table,
go to time-out

          //Object higher than threshold line
          if (state == play_high) begin
                countdown_2 <= countdown_2 + 1;
                if (y_center > threshold_height + buffer + YOFFSET)
begin
                      state <= play_low_af;
                      countdown_2 <= 0;
                end
                else if (countdown_2 == time_out) state <=
record_end_play; //Time-out if it

                                                //doesn't drop below
threshold line
          end

          //Object lower than threshold line
          if (state == play_low_af) begin
                countdown_2 <= countdown_2 + 1;
                if (y_center > table_height + buffer + YOFFSET) state <=
record_end_play;
                else if (countdown_2 == time_out) state <=
record_end_play;  //Below table
          end
          if ((state == record_end_play)&(countdown < point_to_end))
//Time-out
                countdown <= countdown + 1;
          else if ((state == record_end_play)&(countdown ==
point_to_end)) begin
                state <= idle;
                record <= 0;
          end
     end
endmodule
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
module center_of_mass(
       input average,
```

```
    input clk,
    input match,
    input [10:0] hcount,
    input [9:0] vcount,
    output reg [12:0] x_center,
    output reg [11:0] y_center,
       output reg [23:0] final_count, // used to determine if center due
to noise alone
    output reg done
    );

       parameter XOFFSET = 150;
       parameter YOFFSET = 250;
       parameter XMIN = 0 + XOFFSET;
       parameter XMAX = 700 + XOFFSET;
       parameter YMIN = 10 + YOFFSET;
       parameter YMAX = 500 + YOFFSET;

      reg [43:0] x_buffer;
      reg [39:0] y_buffer;
      reg [31:0] x_sum = 0;
      reg [31:0] y_sum = 0;
      reg [23:0] count = 0;
      reg frame_done = 0;

      // Have a little cushion on the edge of the frame
      reg [10:0] X_Start = XMIN;
      reg [10:0] X_End = XMAX;
      reg [9:0] Y_Start = YMIN;
      reg [9:0] Y_End = YMAX;

      // accumulate coordinate values for the center of mass calculation
      always @(posedge clk)begin
            if ((hcount >= X_Start && hcount <= X_End)
                        &&(vcount >= Y_Start && vcount <= Y_End))begin
                  if (hcount==X_Start && vcount==Y_Start)begin
                        x_sum <= 0;
                        y_sum <= 0;
                        count <= 0;
                        frame_done <= 0;
                  end
                  else if (match)begin
                        x_sum <= x_sum + hcount;
                        y_sum <= y_sum + vcount;
                        count <= count + 1;
                  end
            end
            if (frame_done == 1)
                  frame_done <= 0;
            else if (vcount==Y_End && hcount==X_End+50)
                  frame_done <= 1;
      end

      // Calculate center of mass (average X and Y of matches)
      wire [31:0] x_numer = x_sum;
      wire [23:0] x_denom = count;
      wire [31:0] x_quotient;
      wire [15:0] x_fraction;
      wire x_ready;
```

```
      coreGenDivider32 x_divider(.clk(clk),
.dividend(x_numer),.divisor(x_denom), .quotient(x_quotient),
                        .fractional(x_fraction), .rfd(x_ready));

      wire [31:0] y_numer = y_sum;
      wire [23:0] y_denom = count;
      wire [31:0] y_quotient;
      wire [15:0] y_fraction;
      wire y_ready;
      coreGenDivider32 y_divider(.clk(clk), .dividend(y_numer),
.divisor(y_denom), .quotient(y_quotient),
                        .fractional(y_fraction), .rfd(y_ready));

      always @(posedge clk)begin
            if(frame_done && x_ready && y_ready)begin
                  x_buffer <= {x_buffer[32:0], x_quotient[10:0]};
                  y_buffer <= {y_buffer[29:0], y_quotient[9:0]};
                  final_count <= count;
                  done <= 1;
                  // Calculate the average center
                  x_center <= average ? {2'b0, x_quotient[10:0]} :
((x_center<<2) - x_buffer[43:33] + x_quotient[10:0])>>2;
                  y_center <= average ? {2'b0, y_quotient[9:0]} :
((y_center<<2) - y_buffer[39:30] + y_quotient[9:0])>>2;
            end
            else begin // done only asserted for one cycle
                  if (done)
                        done <= 0;
            end
      end

endmodule

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
module ball_generator(
    input [12:0] x_center,
    input [11:0] y_center,
    input [10:0] hcount,
    input [9:0] vcount,
    input [15:0] matches,
    input [17:0] background,
       input replay,
    output reg [17:0] pixel
    );
       parameter WIDTH = 7;
       parameter HEIGHT = 7;
       parameter BLUE = {6'h00,6'h00,6'h3f};
       parameter GREEN = {6'h00,6'h3f,6'h00};
       parameter RED = {6'h3f,6'h00,6'h00};

       always @* begin
            if (!replay)begin
                  if (((hcount+WIDTH) >= x_center && hcount <
(x_center+WIDTH)) &&
                        ((vcount+HEIGHT) >= y_center && vcount <
(y_center+HEIGHT)))
//                      pixel = (matches > 16'h0300) ? GREEN : RED;
                        pixel = GREEN;
                  else
                        pixel = background;
```

```
            end
            else begin
                  if (((hcount+20) >= x_center && hcount < (x_center+20))
&&
                        ((vcount+20) >= y_center && vcount <
(y_center+20)))
                        pixel = BLUE;
                  else
                        pixel = background;
            end
       end

endmodule
```