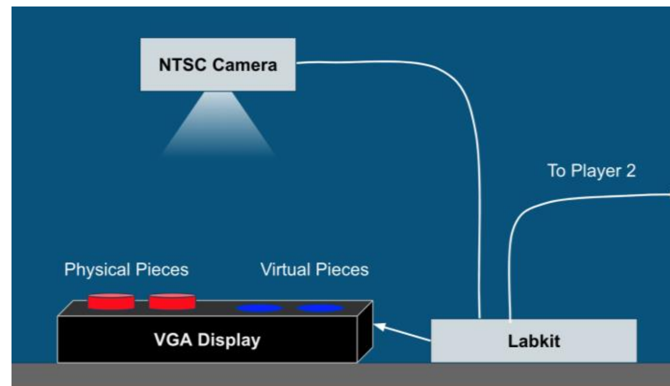
Check Yourself

6.111 Final Project Report
Fall 2018

Suzanne O'Meara
Elijah Stanger-Jones
August Trollbäck

Introduction

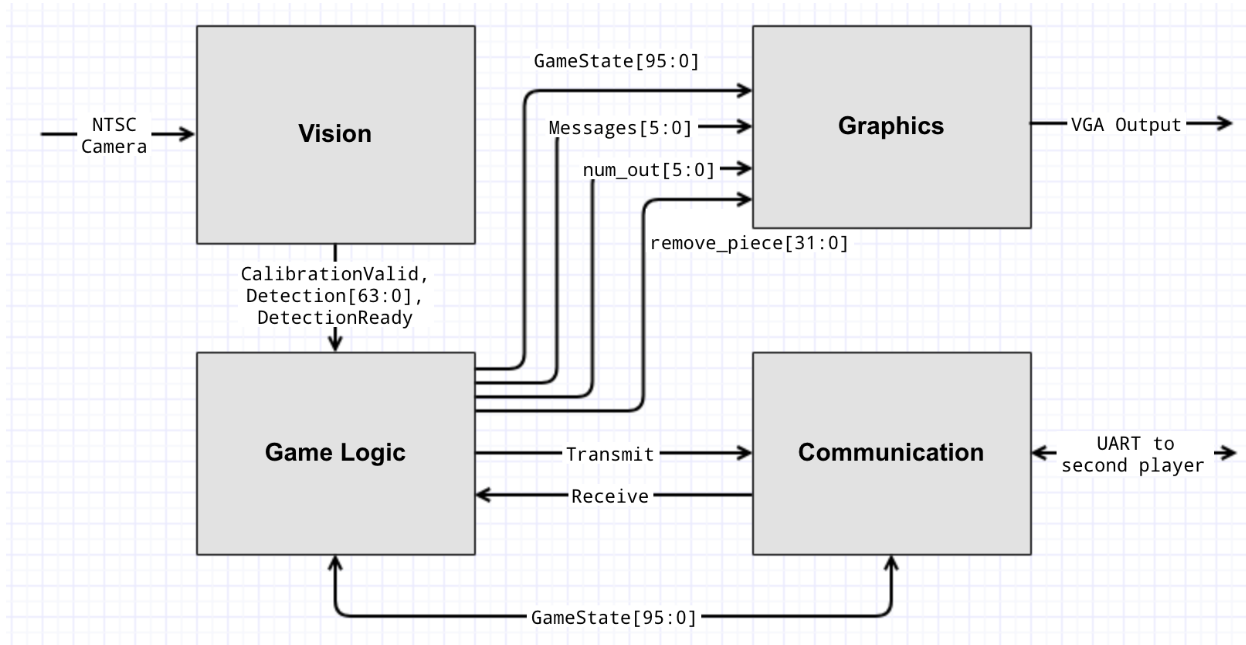
Check Yourself is an FPGA implementation of a multiplayer game of checkers that has a hybrid of virtual and physical gameplay. The game allows two players in different locations to play a game of checkers by moving their physical checker pieces and watch their opponents pieces move virtually in real time. Each player's setup has an FPGA Labkit connected to a monitor laid flat that acts as the checkerboard and a camera viewing the board from above to locate physical checkers placed atop the screen. A RS232 cable is connected between the two player's setups to enable communication between the two FPGAs running identical bitcode.



The FPGA has an internal game logic module that keeps track of the game's progression and enforces that all moves are rules-compliant before allowing to submit their turn to the opponent. The new game state is communicated to the opponent by the communications module that implements redundancy so the system never enters an impossible state. A graphics module displays the current game state to the user, along with messages indicating who's turn it is, whether the player made an illegal move, or if the game has ended.

System Architecture

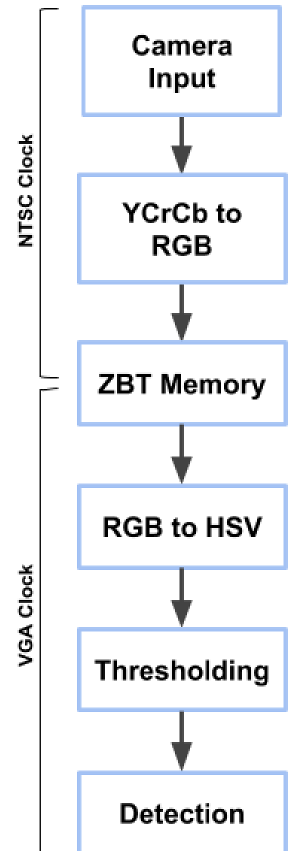
The block diagram shows the high-level organization of *Check Yourself*. Each module is decoupled from the rest and operates through a well-defined interface. This made it possible for each team member to work individually on their subsystem and minimized issues during the integration phase.



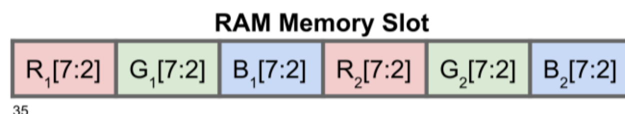
Computer Vision (August)

Implementation

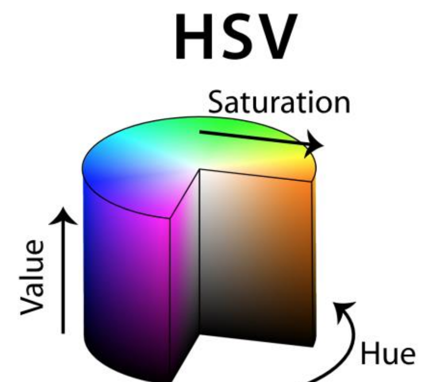
One of the important decisions to make in the computer vision module was whether to place certain image processing operations before or after storing to RAM. I originally began implementing all of the detection code to operate directly off of the NTSC input stream but later decided to run detection on the stored frame in RAM, as seen by the VGA control signals. One reason for this was that finding a mapping from NTSC pixel coordinates to VGA coordinates was more difficult than expected. Another big factor was that all operations that happened before RAM would have to be essentially debugged blind, because the data wasn't readily available to display to VGA. Being able to visually see the results of each step in the computer vision pipeline was crucial.



The final implementation of the computer vision pipeline had the following flow (shown graphically on the right). First, frames from the NTSC camera in YCrCb color space were converted to RGB colorspace using a provided module. To account for the pipelining in the conversion module, a 5 count delay was added to the NTSC data valid `dv` and synchronization signals `fvh[2:0]`. The provided `ntsc2zbt` module was then modified to save two pixels consisting of the upper 6 bits of R, G, B channels in each memory location. It is necessary to share memory like this because an entire frame in 24-bit RGB color would be too large to fit in RAM. The memory addressing scheme was also changed to match have two pixels per slot by including another lower bit of the x address. This picture shows how two consecutive pixels were stored in memory:



After this, all computer vision was done with the frame stored in RAM, using the generated VGA signals from `xvga`. RGB data stored in RAM was extended back into a 24 bit bus and sent to the `rgb2hsv` module to convert to HSV (hue, saturation, value) color space. The conversion takes 22 clock cycles, so VGA signals were delayed appropriately for later use. HSV is a good color space to use for color-based detection because it isolates the true color (hue) from the 'pureness' of the color (saturation) and brightness of the color (value). Unlike the YCrCb to RGB conversion which is a fairly simple linear transform, converting from RGB to HSV is more complicated because HSV is cylindrical coordinate space, hence the longer pipeline delay.



The HSV data was then converted into a binary mask by thresholding on hue, saturation, and value. Thresholding sets each pixel to the boolean result of ANDing whether each HSV channel was inside its given range. A good binarization was found by using the labkit user inputs to increment and decrement the upper and lower bounds for each channel's range. We found that lighting conditions were very important to the quality of the binarized image. Intense point lights (like a phone flash) were not very effective because they washed out colors, removing most of the saturation. More diffused lighting worked much better.

An example image of the HSV thresholding. The binarized mask only includes the red checkers.

Classifying a grid square as having a checker piece by looking at a single pixel in the mask was not robust enough to noise, so a module was made for finding the average density of pixels in a bounding box around the center of a grid square. If the density was above a certain threshold parameter, the grid square was considered to contain a physical checker piece. A density calculation module instantiation was generated for each center location in the 8x8 grid. The detection signals for each square were then passed to the game logic, along with logic to ensure that the data was ready to be consumed. Originally the idea was to have only a single one of these modules instantiated and have it's target location change depending on which grid square the game logic module wanted to check, but it was easy enough to run detection on all the grid squares at once.

A secondary mask was used to find the light green calibration squares displayed on each corner of the checkerboard. Similar to the checker piece detection, a density calculation module was generated for each calibration location. The computer vision module continuously tracked these calibration points and signalled to the game logic module that recalibration was necessary if less than three out of the four calibration markers were properly positioned.

Challenges

One of the most confusing problems I encountered was that the mask generated from thresholding HSV data had 'ghosting' effects, by which I mean there were areas that registered as being a red checker but were actually offset from the actual position of the checker piece. The solution ended up being that the divider IP core module used internally by the RGB to HSV module was generated with the incorrect remainder option (you can choose between remainder and fractional). For some reason this option led to the hue channel being delayed a few more

cycles than the saturation and value channels. Once the divider IP core was generated again with the correct options the computer vision worked much better.

Another implementation detail that took longer than it should have to figure out was that the hue values corresponding to red wrapped around from 0 to 255. So instead of masking hue on using **`hue_lo < hue < hue_hi`**, the range was inverted: **`~(hue_lo < hue < hue_hi)`**. Only looking at the lower half of the red hue's (zero and above) caused the mask to appear very noisy, so I spent a lot of time trying to track down this non-existent noise in other parts of the computer vision module.

Communications (Suzanne)

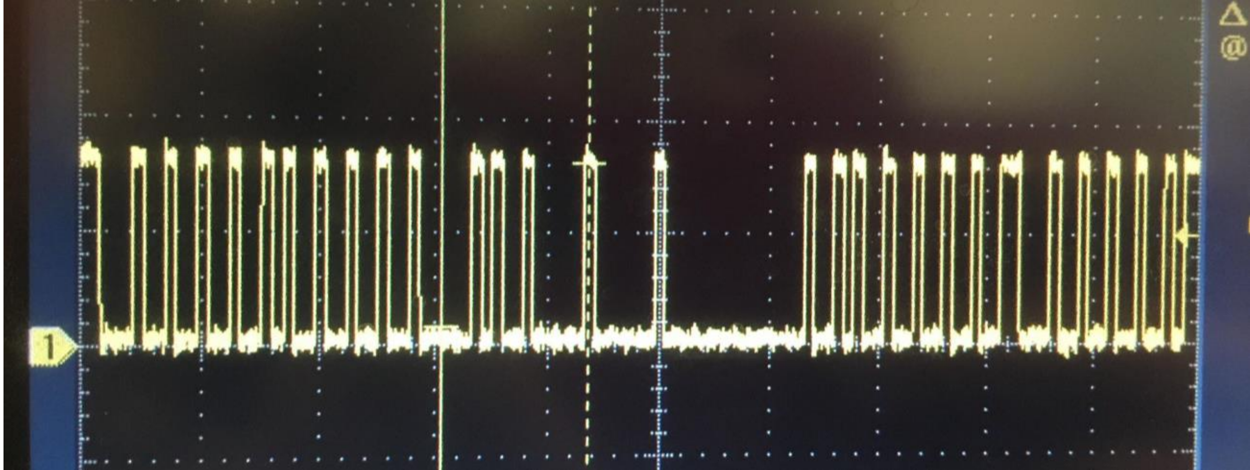
Implementation

In order to communicate between the two FPGAs, I implemented a limited version of UART that would send the full game state stored in one FPGA to the other. This communications scheme was asynchronous, but did rely on both FPGAs having a 65 MHz internal clock. Instead of sending messages in 8 or 16-bit chunks, my communication module sent all 96 bits of data in one message, with a start and stop bit on either end. I judged that this would be an acceptable way to send the data back and forth, because the only type of data we needed to send between the FPGAs was a single type of 96-bit message.

The transmitter module takes in two pieces of information from the game state module: the full game state and a "transmit now" signal. The full game state data bus is 96 bits wide and specifies where each piece is on the checkerboard. The "transmit now" signal is a single, active high signal. Whenever it is high, the transmit module will package the state data into a 127 bit message and send it to the other FPGA through the designated RS232 pins. There are extra bits in this message due to a "data verification" tag that is added to the front end. This tag contains a series of bits that would not be found normally in the message. The transmit module also generates a 28.8 kHz clock signal that dictates the baud rate for the UART bus. The transmit module sends information by shifting one bit at a time out of the stored 127 bit message to the RS232 TX pin on the FPGA labkit.

Start bit	Verification Tag	Data [96 bits]	Stop bit

This image shows the format of each of the messages sent over our modified UART



This image shows us an oscilloscope screenshot of the data section of the UART transmission

The transmit module contains some noise-rejection measures. Every time that it sees the “transmit now” signal go high, it will send the same exact message to the RS232 TX pin 50 times. Each of these 50 messages is separated from the next by a dead time that is longer than each message. After the module sends the 50th message, it goes back to waiting for a new transmit signal. If the transmit signal is still high, it will send another 50 messages. This allows the checkers player to resend the game state as many times as they would like, which is useful in the scenario that all 50 messages were too noisy for the receive module to read.

The receive module contains the same 28.8 kHz clock signal as the transmit module. The receive module is always looking at the state of the dedicated RS232 RX pin on the FPGA labkit. That trace is always high when there is no data available. When the receive module sees that signal go low, it interprets it as a start bit and begins the process of verifying the start bit and checking that the rest of the data is valid.

The receive module verifies that a start bit is real by sampling the RX trace from the RS232 connector 30 times in one half-period of the UART clock. If more than 3 of those samples are read as high, then the start bit is deemed “fake” or too noisy. The receive module would then go back to its start state and wait for the length of one message before looking for another start bit. The module idles for the length of a full message just in case the start bit was “real” and is followed by a full message. In this case, you would not want the receive module to keep thinking every low bit of this message is another start bit.

Once the receive module determines that a start bit is “real”, it verifies that the data is not noisy. It reads the value of the verification tag at the beginning of the message, and compares it to a stored value. Each message has the same verification tag attached, so the receive module can compare it to the same stored value every time. If the tag does not match the stored value, the message is deemed “too noisy” and the state machine will loop back to its waiting state. It will again wait in this state for the length of one message before looking for another start bit. This is safe to do, because the transmit module inserts over one message’s worth of dead time between messages, so the receive module will not be missing the beginning of another message while it waits.

Once the receive module has recorded a full message, it splices out the game state data and repackages it into the 96-bit game state bus. The game state module has access to this bus all the time. In order to tell the game state module that the data is ready, the receive module writes a high value to the "message received" wire going out of the receive module. The game state module is waiting for this "message received" signal and will then record the 96-bit data bus.

Challenges

One of the main challenges I had while writing and testing the communications modules was the presence of noise on the wires that were used to transmit the data. This noise often caused the receive module to read data bits that did not accurately reflect the data that was sent by the transmit module. This problem often manifested as the receive module missing a start bit or reading a start bit where there was none, and then returning a bogus state to the game state module. This would cause the graphics module to display weird and wrong combinations of checkers on the checkerboard.

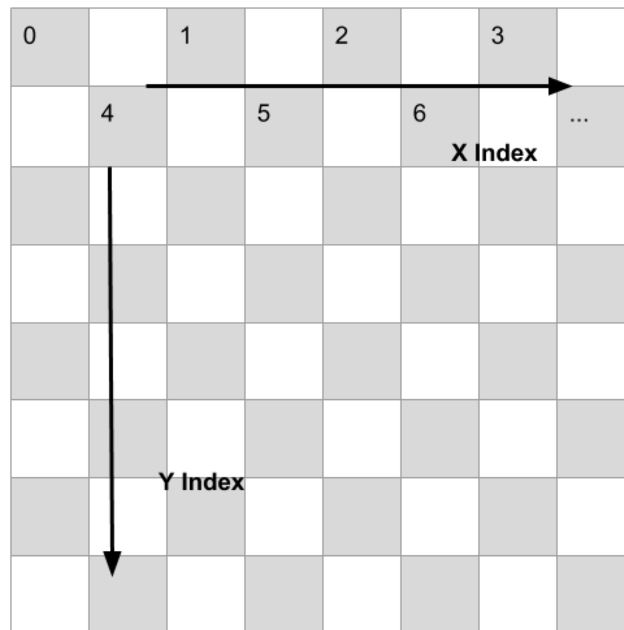
This was a difficult issue to deal with because it happened very intermittently. We would often play over a dozen moves on a checkers game and then one transmission would go wrong and ruin the entire display and game state. In order to debug the issue, I had to isolate the UART code and test it separately. The fixes I implemented are described above, but specifically involved sending each message many times, verifying start bits more carefully, and adding a data validation tag at the beginning of each message.

Graphics (Suzanne)

Implementation

The graphics for this project were initially very function-oriented. They needed to clearly convey to the game's user where each piece was on the checkerboard, and display any relevant messages that would tell the user what to do next. Once the graphics modules had accomplished this much, I focused on making the graphics more aesthetically pleasing and similar to playing checkers in real life.

The first piece of the graphics that I implemented was the checkered game board. To make this easy to display, I chose to have each box on the board be 64 by 64 pixels. This meant that in order to determine which square on the board a particular pixel was in, I could just shift the **`hcount`** and **`vcount`** values by 7 and 6 to the right and would get the x or y index of that pixel on the game board. The color of each of the pixels on the game board was determined by dividing the **`hcount`** and **`vcount`** values by 128 and taking the modulus. The only modification I made to this scheme was to center the board on the screen. To do this, I just subtracted a constant offset from **`hcount`** and **`vcount`** before generating the game board.



*This shows the checkerboard. Each piece on the board had an x index, a y index, and an absolute index (the number displayed in some squares). The absolute index was calculated as (X_index + 8*Y_index).*

The next part of the graphics module that I implemented was the game pieces. The game piece display was determined by the 96-bit data bus that the graphics module receives from the game state module. The bus is formatted such that each group of adjacent bits in a row reflects the state of a single square on the checkerboard. If the first bit is high there is a red piece, if the second bit is high there is a blue piece, and if the third bit is high there is a king (green). If all of the bits are 0, the board space is blank. In order to translate this data into pieces on the board, the graphics module first determines which square on the board it should be in based on its offset hcount and vcount values. From there, it can calculate the index of that square in the 96-bit bus, and determine if there should be a piece present in the checkers square that that pixel is a part of. If there is, it assumes that there will be a 25-pixel radius circle centered in that checkers square. If it determines that the pixel is within the radius of that circle, the graphics module will turn it the color of the correct game piece. If it is not within the radius, that pixel will default to the background color of the checkerboard.

One part of the piece graphics that I implemented later was adding pictures to the displayed pieces. Most of the implementation of the picture-pieces was the same as for solid-color pieces. The only change happens once the pixel has been determined to be within the circular bounds of the checker piece. Then, instead of just assigning that pixel the color of the piece, it is assigned the value of the output of a **`picture_blob`** module. The blob module is part of the graphics module family, and can display bitfiles at an arbitrary location on the video screen. I will describe it in more detail below. Different pictures were displayed on each of the pieces to indicate whether a piece was a regular checker or a king.

Left: the full-color version of the dog displayed on "regular" pieces. Right: the full-color version of the frog used to indicate when an opponent's piece became a king.

Another job of the graphics module was to display the pieces that had been jumped by the other player on the side of the checkerboard. In order to display these pieces, the graphics module receives a count from the game state module that says how many game pieces had been captured by the other player. The graphics module reserves a grid of spaces next to the game board where the "out" pieces could be displayed. The mechanics of displaying these pieces is essentially identical to how the pieces on the board were displayed. The only difference was how the module determined whether or not a piece should be present at a particular grid location. Instead of checking the 96-bit bus, the module would compare the index of that grid location to the total number of pieces that had been captured by the other player. If the index was smaller than the number of pieces that were out, then a piece would be displayed there.

The graphics module also indicates when a physical piece should be removed by the user. It indicates this by causing the square under that piece to flash yellow until the user removes the piece and presses enter.

The second most critical job of the graphics module was to display messages from the game state to the user about what they should do next. These messages included phrases such as "Your Turn", "You win!", and "Illegal Move." I implemented these messages by making a unique bitfile for each message in MS Paint. I then created a ROM on the FPGA for each message, and a module called **`picture_blob`** to access them. The picture blob module takes in pipelined versions of **`hcount`** and **`vcount`**, the desired x and y coordinates of the upper left corner of the picture, and a bus indicated which message should be displayed. This was possible to implement with only one copy of this **`picture_blob`** module, because only one message had to be displayed at a time. Within the picture blob module there was an instance of each of the rom modules, which were all generating the appropriate pixel value in parallel. The pixel for the desired message would then be selected as the output pixel based on the value of the message select bus. As mentioned above, another instance of this module was used to add a picture to each of the checker pieces on the board. Only one instance of this module was

necessary to generate up to 64 pictures (one for each checker piece), because the x and y coordinate inputs to the **`picture_blob`** module could be changed in between displaying each picture.

Three different messages that were used to convey game-play information to the each player.

<u>Challenges</u>

I found that different tasks required to display certain parts of the graphics on the screen took the FPGA different amounts of time. This meant that some of the tasks, such as displaying the game board, could be accomplished within a single system clock cycle and therefore required no pipelining. Other tasks, such as fetching bits from ROM for the message display or checking whether a pixel was within the radius of a circle, required too much time for one clock cycle and had to use pipelined **`hcount`**, **`vcount`**, and output signals.

The process of integrating with other modules of the game revealed more bugs within the graphics module. Adding more modules outside the graphics caused some of the graphics processes to take longer, so they had to be pipelined more aggressively. This caused some difficulty because I had to keep editing the pipelining for certain parts of the graphics module long after I had thought it was finished. This was especially challenging because I was continuing graphics development in parallel with system development, but on a different machine. I did this because towards the end of system integration, our compile times exceeded 6 minutes. Given that graphic changes were often made incrementally, this made it extremely difficult to efficiently identify and eliminate bugs in my code. Therefore I made changes on a separate machine and pipelined more aggressively in an attempt to preempt any latency issues resulting from integration.

Game Control Module (Elijah)

<u>Implementation</u>

The first step in creating the game logic module was to define a data structure to use for both internal calculations and communicating with the UART and Graphics modules. By defining the board squares with the numbering system as above, where the 0 square is the position closest to the player we then used a 96 bit bus to fully define the state of the board which had the following format:

```
game_state[95:0] = {n31, n30, …. n1, n0 }
                 n? = {player1, player2, king}
```

Each board position has a value that indicates whether player1 or player2 has a piece on there, and a third bit which indicates if the piece located there is a king piece.

Transmit Logic

When a player presses the transmit button the system requests the latest computer vision information and this comes in as the following structure:

```
player1[31:0] = {n31, n30, …. n1, n0 }
```

This is then combined into the state information from the last state to create a new state. This leaves us with two 96 bit states: `currentState, previousState.`
The next step is to work out what changed between these two states, to do this we do the following math to create a clear value of where the piece was and where the piece is:

```
diff_state[95:0] = previous_state[95:0] ^ current_state[95:0]
old_position[95:0] = diff_state & previous_state
new_position[95:0] = diff_state & current_state
```

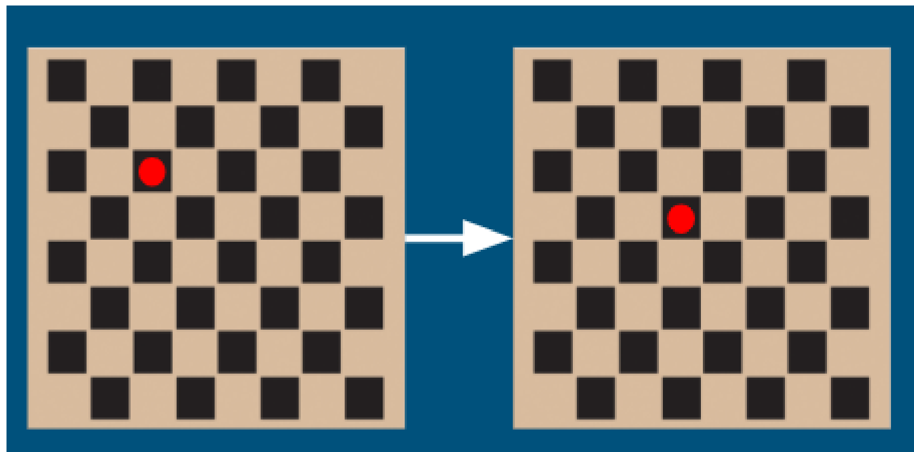
We then convert the two variables into easier to manage integers ranging from 0 -> 31, to make the logic easier for the rest of the state machine.

The state machine then runs through a series of checks before it declares a move valid or invalid. The checks are performed in a specific order as to guarantee that all edge cases are caught and no false positives or worse false negatives occur.

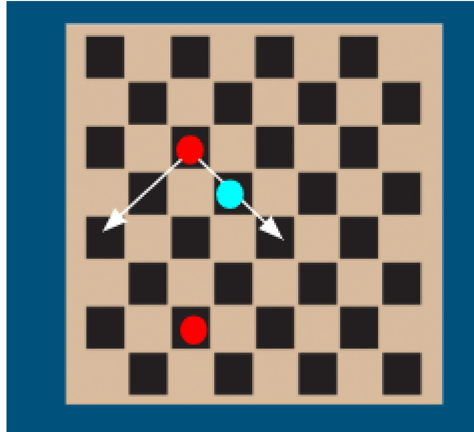
Checks: 1.One piece

- The easiest check in the system is that one and only one piece moved, the fastest way to complete this was to check that only 1 bit was high in both the old position and new position.

2. *Same spot*
   - Check that no board has a piece from both players.
3. *Simple check*
   - Determine if the player has made a legal non jump move.

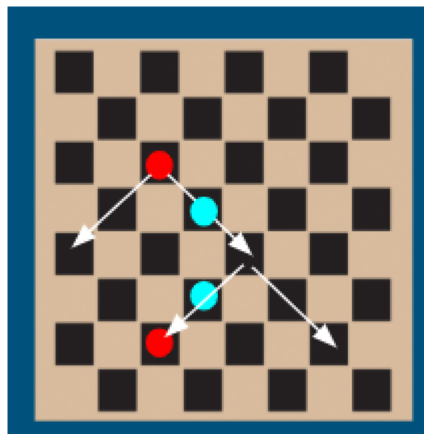


   - There are 3 types of positions: edge positions, odd rows and evens rows.
   - For edge positions the valid location is +4
   - For odd rows the valid locations are +5 and +4
   - For even rows the valid locations are +4 and +3
4. *Jump check 1*
   - For jump check 1, the process is the same but this time with an extra check to make sure an opponent piece exists between the two positions.



- For this check there are 4 types of positions: edge left, edge right, odd rows, even rows.
- For edge left, the valid position is +7 with the opponent at + 4
- For edge right, the valid position is +9 with the opponent at +4
- For odd rows, the valid position is +9 with the opponent at +5, or + 7 with the opponent at +4
- For even rows, the valid position is +9 with the opponent at +4, or +7 with the opponent at +3
- If a valid position is found the opposition piece is removed from the state value and the system transmits to the opposition.

5. *Jump check 2*
   - For jump check 2, the process is the same but this time there is a check for 2 opposition pieces and an empty space.



- For this check there are significantly more cases to check for, these can all be seen in the Verilog code attached in the appendix. I will discuss later on why this increased dramatically and how if we had had this insight at the beginning we could have avoided this.

If the system fails at a initial test (same spot, one piece) or doesn't successful pass any of the position checks the user is informed that the move was invalid and the system returns to waiting for the transmit button to be pressed.

If the system passes we run a check to see if the game is over (all of the opposition pieces have been removed) and then transmit the current game state to the UART system.

Receive Logic

The receive logic is a lot more basic, with the main purpose to reverse the bits so that the logic stays the same regardless of whether you are player1 or player2. This required the bits in the location of player1 to be switched with the bits in the location of player2 and for the board to be flipped around. The verilog for this process is below:

```
  current_state[i*3] = uart_state[(31-i)*3]; //king
current_state[1+i*3] = uart_state[2+(31-i)*3]; //p2
current_state[2+i*3] = uart_state[1+(31-i)*3]; //p1
```

The second step is to detect if a) the game has ended and b) if you need to remove any pieces. Detecting if the game has ended is a simple process, we check the state to see if the player has no pieces remaining and if so declare them the loser.

To detect if pieces need to be removed we do a simple XOR process from the transmit section and then determine which pieces are missing. We then send this information to the graphics section to indicate to the user which piece to remove. Once the users presses the button we run another check with updated info from the CV module to check that the piece has actually been removed before returning to the waiting state at the top of the state machine.

King Logic

The main stretch goal for the game module was to implement the logic for king pieces, this obviously adds another level of complexity when checking for legal moves.
The first step is that anytime a piece moves into the last line assigning that piece as a king piece. Then we simply add a few extra levels of checks in the state machine for if the piece has the king bit active. Again because of the way we had defined the axis of the checkers board this was a very complicated addition.
For the simple check we had to add the following valid positions:
Edges: +4, -4
Odd rows: +5, +4, -4, -3
Even rows: +4, +3, -5, -4

For the jump checks there was a more complicated addition which can be viewed in the verilog code.

Challenges

The first challenge was due to the sheer number of possible unique moves in a game of checkers, its very difficult to guarantee that you reached all edge cases without significant testing. One way that I could've made this processor easier would've been to write an increased number of test cases and more complicated test cases to run before the integration process.

The second challenge was due to an early infrastructure decision we made regarding the way to define the location of each piece. Inherently as humans we view a checkers board as a square, but as it turns out because of the every second square system of checkers defining the system in an x,y axis means that a lot of the math was not symmetric. This lead to every level of depth check increasing in complexity. A better solution to this would've been to define the board on a diagonal based system.

The third challenge was that testing before the whole system was integrating was difficult, writing test cases for hundreds of different moves wasn't reasonable and likely wouldn't have caught many bugs anyway. Once the system was integrated it was much easier to test by playing over and over again, finding edge cases that I'd missed quite regularly.

Lessons Learned & Advice

- Invest some time in debugging tools/test benches. It sometimes seems that the upfront cost of writing debugging tools isn't worth it, but with the growing compile times from integration complexity you can save a lot of time in the long run. For instance originally the HSV thresholding ranges were hardcoded – because the ambient lighting changes throughout the day it was necessary to constantly change these values. It ended up being very helpful to make the values modifiable during run-time.
- Use the logic analyzer! Especially for debugging systems that use buses like camera/graphics signals. This was useful for tracking down some tricky issues where NTSC data was accessed when the data wasn't valid yet.
- Configure your graphics code to be easily pipelined early. When the latency of certain actions varies during development, this will make it easy to fix your graphics.
- Rigorously test any communications-related code and hardware before integrating it with the rest of the system. Communications bugs can look a lot like bugs in other parts of the system.

- Test benches for every subsystem will save you a lot of time and headache, especially any systems that are doing a lot of math like our main state machine. With a testbench it becomes a lot easier to see exactly where the math went wrong.

Appendix / Verilog Source

---

calibrate_display.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:23:07 11/18/2018
// Design Name:
// Module Name:    graphics
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module calibrate_display(
   input vclock, // 65MHz clock
   input [10:0] hcount,   // horizontal index of current pixel (0..1023)
   input [9:0]   vcount, // vertical index of current pixel (0..767)
   input [10:0] width,
   input [9:0] height,
   input [10:0] board_size,
   input display,
   output reg [23:0] cd_pixel     // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );

   wire [11:0] X1;
   assign X1     = ((width - board_size)/2) - 40;
   wire [11:0] X2;
   assign X2 = ((width - board_size)/2) - 10;
   wire [11:0] X3;
   assign X3     = ((width + board_size)/2) + 10;
   wire [11:0] X4;
   assign X4     = ((width + board_size)/2) + 40;
   wire [11:0] Y1;
   assign Y1     = ((height - board_size)/2) - 40;
   wire [11:0] Y2;
   assign Y2     = ((height - board_size)/2) - 10;
   wire [11:0] Y3;
   assign Y3     = ((height + board_size)/2) + 10;
```

```
   wire [11:0] Y4;
   assign Y4     = ((height + board_size)/2) + 40;

   reg [23:0] temp_pixel;

 always @(*) begin
    temp_pixel = 24'h00_00_00;
     if (display) begin
         if (((hcount>X1)&(hcount<X2))|((hcount>X3)&(hcount<X4))) begin
             if (((vcount>Y1)&(vcount<Y2))|((vcount>Y3)&(vcount<Y4))) begin
                 temp_pixel = 24'h00_FF_00;
             end
         end
     end
     cd_pixel = temp_pixel;
 end

endmodule
```

calibration_drawing.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:10:12 12/02/2018
// Design Name:
// Module Name:    calibration_drawing
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module calibration_box(
    input clk,
    input [10:0] hcount,
    input [9:0] vcount,
    output hit
    );

        parameter THICKNESS = 10;
        parameter X1 = 11'd100;
        parameter X2 = 11'd400;
        parameter Y1 = 10'd100;
        parameter Y2 = 10'd400;

        wire inner_box = (hcount >= X1 & hcount <= X2 & vcount >= Y1 & vcount <= Y2);
        wire outer_box = (hcount >= (X1-THICKNESS) & hcount <= (X2+THICKNESS) & vcount >= (Y1-THICKNESS) & vcount
<= (Y2+THICKNESS));
        assign hit = outer_box & ~inner_box;

endmodule

module draw_bbox(clk, hcount, vcount, x, y, hit);
        input clk;
        input [10:0] hcount, x;
        input [9:0] vcount, y;
        output hit;

        parameter RADIUS = 4;

        assign hit = (x-RADIUS < hcount) & (x+RADIUS > hcount) & (y-RADIUS < vcount) & (y+RADIUS > vcount);
endmodule
```

comms.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:21:18 11/18/2018
// Design Name:
// Module Name:    comms
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module transmit(
    input [95:0] state_in,
    input transmit,
         output tx,    // serial data sent to RS232 output driver
         input clk,
    output reg xmit_clk,  // baud rate; sent to logic analyzer for debugging
    );

         // this section sets up the clk;
    parameter DIVISOR =  234; // create 115,200 baud rate clock, not exact, but should work.
    parameter MARK = 1'b1;
    parameter STOP_BIT = 1'b1;
    parameter START_BIT =1'b0;

    reg [15:0] count;

    always @ (posedge clk)
    begin
       count <= xmit_clk ? 0 : count+1;
       xmit_clk <= count == DIVISOR-1;
    end

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

        reg [95:0] state_st;
        reg [108:0] big_data_tx;
        reg [108:0] buff1;
        reg [108:0] buff2;
        reg [108:0] buff3;
        reg [11:0] t_count;
        reg [5:0] send_count;
        //reg transmit_on = 0;

        /*always @(posedge clk)
        begin
                //transmitting
           if ((transmit == 1) & (transmit_on == 0)) begin
```

```
                        transmit_on <= 1;
                        t_count <= 0;
                        big_data_tx <= {1'b1, state_in[95:80], 2'b01, state_in[79:64], 2'b01, state_in[63:48],
2'b01, state_in[47:32], 2'b01, state_in[31:16], 2'b01, state_in[15:0], 2'b01};
                end
                else if (t_count >= 200) begin
                        transmit_on <= 0;
                end
                else begin
                        if (count == 1) begin
                                big_data_tx <= {1'b1, big_data_tx[108:1]};
                                t_count <= t_count + 1;
                        end
                end
        end */

        always @(posedge clk)
        begin
                //transmitting
           if (transmit == 1) begin
                        state_st <= state_in;
                        send_count <= 0;
                        t_count <= 0;
                end
                if (t_count >= 200) begin
                        //if (send_count < 10) begin
                        big_data_tx <= {1'b1, state_st[95:80], 2'b01, state_st[79:64], 2'b01, state_st[63:48],
2'b01, state_st[47:32], 2'b01, state_st[31:16], 2'b01, state_st[15:0], 2'b01};
                        send_count <= send_count + 1;
                        //end
                        t_count <= 0;
                end
                else begin
                        if (count == 1) begin
                                big_data_tx <= {1'b1, big_data_tx[108:1]};
                                t_count <= t_count + 1;
                        end
                end
        end

        assign tx = big_data_tx[0];

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

endmodule
```

datboi_rom.v

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file datboi_rom.v when simulating
// the core, datboi_rom. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module datboi_rom(
        clka,
        addra,
        douta);


input clka;
input [11 : 0] addra;
output [7 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
                .C_ADDRA_WIDTH(12),
                .C_ADDRB_WIDTH(12),
                .C_ALGORITHM(1),
                .C_BYTE_SIZE(9),
                .C_COMMON_CLK(0),
```

```
        .C_DEFAULT_DATA("0"),
        .C_DISABLE_WARN_BHV_COLL(0),
        .C_DISABLE_WARN_BHV_RANGE(0),
        .C_FAMILY("virtex2"),
        .C_HAS_ENA(0),
        .C_HAS_ENB(0),
        .C_HAS_MEM_OUTPUT_REGS_A(0),
        .C_HAS_MEM_OUTPUT_REGS_B(0),
        .C_HAS_MUX_OUTPUT_REGS_A(0),
        .C_HAS_MUX_OUTPUT_REGS_B(0),
        .C_HAS_REGCEA(0),
        .C_HAS_REGCEB(0),
        .C_HAS_SSRA(0),
        .C_HAS_SSRB(0),
        .C_INIT_FILE_NAME("datboi_rom.mif"),
        .C_LOAD_INIT_FILE(1),
        .C_MEM_TYPE(3),
        .C_MUX_PIPELINE_STAGES(0),
        .C_PRIM_TYPE(1),
        .C_READ_DEPTH_A(3800),
        .C_READ_DEPTH_B(3800),
        .C_READ_WIDTH_A(8),
        .C_READ_WIDTH_B(8),
        .C_SIM_COLLISION_CHECK("ALL"),
        .C_SINITA_VAL("0"),
        .C_SINITB_VAL("0"),
        .C_USE_BYTE_WEA(0),
        .C_USE_BYTE_WEB(0),
        .C_USE_DEFAULT_DATA(0),
        .C_USE_ECC(0),
        .C_USE_RAMB16BWER_RST_BHV(0),
        .C_WEA_WIDTH(1),
        .C_WEB_WIDTH(1),
        .C_WRITE_DEPTH_A(3800),
        .C_WRITE_DEPTH_B(3800),
        .C_WRITE_MODE_A("WRITE_FIRST"),
        .C_WRITE_MODE_B("WRITE_FIRST"),
        .C_WRITE_WIDTH_A(8),
        .C_WRITE_WIDTH_B(8),
        .C_XDEVICEFAMILY("virtex2"))
inst (
        .CLKA(clka),
        .ADDRA(addra),
        .DOUTA(douta),
        .DINA(),
        .ENA(),
        .REGCEA(),
        .WEA(),
        .SSRA(),
        .CLKB(),
        .DINB(),
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());
```

```
// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of datboi_rom is "black_box"

endmodule
```

detection.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    22:03:18 12/02/2018
// Design Name:
// Module Name:    bbox_counter
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module vga_signal_delay(clk,
                                hcount_in, vcount_in, hsync_in, vsync_in,
                                hcount_out, vcount_out, hsync_out, vsync_out);
    parameter DELAY = 3;

    input          clk;
    input [10:0]  hcount_in;
    input [9:0]   vcount_in;
    input          hsync_in;
    input          vsync_in;
    output [10:0] hcount_out;
    output [9:0]  vcount_out;
    output         hsync_out;
    output         vsync_out;

    reg [10:0] hcount [DELAY-1:0];
    reg [9:0]  vcount [DELAY-1:0];
    reg [DELAY-1:0] hsync;
    reg [DELAY-1:0] vsync;

    integer i;
    always @(posedge clk) begin
        for (i=(DELAY-1); i>0; i=i-1) begin
            hcount[i] = hcount[i-1];
            vcount[i] = vcount[i-1];
            hsync[i]  = hsync[i-1];
            vsync[i]  = vsync[i-1];
        end
        hcount[0] = hcount_in;
        vcount[0] = vcount_in;
        hsync[0]  = hsync_in;
        vsync[0]  = vsync_in;
    end

    assign hcount_out = hcount[DELAY-1];
    assign vcount_out = vcount[DELAY-1];
    assign hsync_out = hsync[DELAY-1];
```

```
    assign vsync_out = vsync[DELAY-1];

endmodule

module bbox_counter(
    input clk,
    input vsync,
    input [10:0] x,
    input [9:0] y,
    input [10:0] target_x,
    input [9:0] target_y,
    input pixel,
    input start_cv_in,
     output reg ready_out,
    output reg [7:0] count
    );

    reg [7:0] internal_count = 0;

    parameter RADIUS = 4;

    reg old_vsync;
    reg old_start_cv_in;
    wire vsync_edge = vsync & ~old_vsync;
    wire start_cv_edge = start_cv_in & ~old_start_cv_in;
    reg frame_valid = 1;
    always @(posedge clk) begin
        if (vsync_edge) begin
            count <= internal_count;
            internal_count <= 0;
            frame_valid <= 1;
            if (frame_valid) begin
                ready_out <= 1;
            end
        end
        else ready_out <= 1;

        if (start_cv_edge) begin
            frame_valid <= 0;
        end

        if ( (target_x+RADIUS>x) & (target_x-RADIUS<x) &
                (target_y+RADIUS>y) & (target_y-RADIUS<y) ) begin

            internal_count <= internal_count + pixel;
        end

        old_vsync <= vsync;
    end

endmodule

module detector(clk, vsync,
                        x_rgb, y_rgb, x_hsv, y_hsv, x_grid, y_grid,
                        mask_pixel, mask_cal_pixel,
                        detect_array_out, indicator,
                        debug_counter,
                        start_cv_in, ready_out,
                        calib_good);
    input clk, vsync;
```

```
input [10:0] x_rgb, x_hsv;
input [9:0] y_rgb, y_hsv;
input [7:0] x_grid, y_grid;
input mask_pixel, mask_cal_pixel;
output [63:0] detect_array_out;
output indicator;
output [7:0] debug_counter;
input start_cv_in;
output ready_out;
output calib_good;

parameter LL_X = 11'd200;
parameter LL_Y = 10'd200;
parameter STRIDE = 37;
parameter STRIDE_DIV2 = 18;
parameter CAL_MARKER_OFFSET = 33;
parameter SEARCH_RADIUS = 5;
parameter COUNT_THRESH = 20;
parameter COUNT_THRESH_CAL = 13;

/*
wire [7:0] count;
wire [10:0] target_x = LL_X + STRIDE*x_grid;
wire [9:0]  target_y = LL_Y + STRIDE*y_grid;
assign debug_counter = count;
bbox_counter #(.RADIUS(5)) _bbox_ctr(.clk(clk), .vsync(vsync),
                                .x(x_hsv), .y(y_hsv),
                                .target_x(target_x), .target_y(target_y),
                                .pixel(mask_pixel),
                                .count(count),
                                .start_cv_in(start_cv_in), .ready_out(ready_out));
assign detected = (count >= COUNT_THRESH);
*/

wire[63:0] detect_array;
assign detect_array_out = detect_array;
wire [63:0] indicator_array;
wire indicator_accum = |indicator_array;
wire [63:0] ready_out_array;
assign ready_out = &ready_out_array;
genvar i, j;
generate
    for (i=0; i<8; i=i+1) begin : gen_bbox_counters1
        for (j=0; j<8; j=j+1) begin : gen_bbox_counters2
            wire [7:0] box_ctr;
            assign detect_array[i*8 + j] = (box_ctr >= COUNT_THRESH);
            bbox_counter #(.RADIUS(5)) _bbox_ctr_gen(
                .clk(clk), .vsync(vsync),
                .x(x_hsv), .y(y_hsv),
                .target_x(LL_X + STRIDE*i),
                .target_y(LL_Y + STRIDE*j),
                .pixel(mask_pixel),
                .count(box_ctr),
                .start_cv_in(start_cv_in), .ready_out(ready_out_array[i*8 + j])
                );

            draw_bbox #(.RADIUS(4)) _draw_bbox_gen(
                .clk(clk),
                .hcount(x_rgb), .vcount(y_rgb),
                .x(LL_X + STRIDE*i),
```

```
                    .y(LL_Y + STRIDE*j),
                    .hit(indicator_array[i*8 + j])
                    );
            end
        end
    endgenerate

    wire [3:0] cal_indicator_array;
    wire [3:0] cal_detect_array;
    // lets consider 3 points to be good enough
    assign calib_good = (cal_detect_array==4'b0111) |
                            (cal_detect_array==4'b1011) |
                            (cal_detect_array==4'b1101) |
                            (cal_detect_array==4'b1110) |
                            (cal_detect_array==4'b1111);


    wire cal_indicator = |cal_indicator_array;
    generate
        for (i=0; i<2; i=i+1) begin : gen_bbox_cal1
            for (j=0; j<2; j=j+1) begin : gen_bbox_cal2
                wire [7:0] box_ctr;
                assign cal_detect_array[i*2 + j] = (box_ctr >= COUNT_THRESH_CAL);
                bbox_counter #(.RADIUS(5)) _bbox_ctr_gen_(
                    .clk(clk), .vsync(vsync),
                    .x(x_hsv), .y(y_hsv),
                    .target_x(LL_X - CAL_MARKER_OFFSET + (STRIDE*7+CAL_MARKER_OFFSET*2)*i),
                    .target_y(LL_Y - CAL_MARKER_OFFSET + (STRIDE*7+CAL_MARKER_OFFSET*2)*j),
                    .pixel(mask_cal_pixel),
                    .count(box_ctr),
                    .start_cv_in(start_cv_in) /* ignore ready out */);

                draw_bbox #(.RADIUS(4)) _draw_bbox_gen_(
                    .clk(clk),
                    .hcount(x_rgb), .vcount(y_rgb),
                    .x(LL_X - CAL_MARKER_OFFSET + (STRIDE*7+CAL_MARKER_OFFSET*2)*i),
                    .y(LL_Y - CAL_MARKER_OFFSET + (STRIDE*7+CAL_MARKER_OFFSET*2)*j),
                    .hit(cal_indicator_array[i*2+j]) );
            end
        end
    endgenerate

/*  wire bbox_indicator;


    draw_bbox #(.RADIUS(4)) _draw_bbox(.clk(clk),
                            .hcount(x_rgb), .vcount(y_rgb),
                            .x(target_x), .y(target_y),
                            .hit(bbox_indicator));
*/
    wire cbox_indicator;
    calibration_box #(.THICKNESS(5),
                        .X1(LL_X-STRIDE_DIV2), .Y1(LL_Y-STRIDE_DIV2),
                        .X2(LL_X+STRIDE*7+STRIDE_DIV2), .Y2(LL_Y+STRIDE*7+STRIDE_DIV2))
                      _cal_box(.clk(clk),
                                    .hcount(x_rgb), .vcount(y_rgb),
                                    .hit(cbox_indicator));
    assign indicator = cbox_indicator | indicator_accum | cal_indicator;

endmodule
```

display_16hex.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data,
                disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out);

   input reset, clock_27mhz;    // clock and reset (active high reset)
   input [63:0] data;            // 16 hex nibbles to display

   output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
          disp_reset_b;

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Display Clock
   //
   // Generate a 500kHz clock for driving the displays.
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [4:0] count;
   reg [7:0] reset_count;
   reg clock;
   wire dreset;

   always @(posedge clock_27mhz)
     begin
        if (reset)
          begin
             count = 0;
             clock = 0;
          end
        else if (count == 26)
          begin
             clock = ~clock;
             count = 5'h00;
          end
        else
```

```
        count = count+1;
    end

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

////////////////////////////////////////////////////////////////////////////
//
// Display State Machine
//
////////////////////////////////////////////////////////////////////////////

reg [7:0] state;              // FSM state
reg [9:0] dot_index;          // index to current dot being clocked out
reg [31:0] control;           // control register
reg [3:0] char_index; // index of current character
reg [39:0] dots;              // dots for a single digit
reg [3:0] nibble;             // hex nibble of current character

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock)
  if (dreset)
    begin
       state <= 0;
       dot_index <= 0;
       control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
      8'h00:
        begin
           // Reset displays
           disp_data_out <= 1'b0;
           disp_rs <= 1'b0; // dot register
           disp_ce_b <= 1'b1;
           disp_reset_b <= 1'b0;
           dot_index <= 0;
           state <= state+1;
        end

      8'h01:
        begin
           // End reset
           disp_reset_b <= 1'b1;
           state <= state+1;
        end

      8'h02:
        begin
           // Initialize dot register (set all dots to zero)
           disp_ce_b <= 1'b0;
           disp_data_out <= 1'b0; // dot_index[0];
```

```
          if (dot_index == 639)
            state <= state+1;
          else
            dot_index <= dot_index+1;
        end

      8'h03:
        begin
          // Latch dot data
          disp_ce_b <= 1'b1;
          dot_index <= 31;           // re-purpose to init ctrl reg
          disp_rs <= 1'b1; // Select the control register
          state <= state+1;
        end

      8'h04:
        begin
          // Setup the control register
          disp_ce_b <= 1'b0;
          disp_data_out <= control[31];
          control <= {control[30:0], 1'b0};  // shift left
          if (dot_index == 0)
            state <= state+1;
          else
            dot_index <= dot_index-1;
        end

      8'h05:
        begin
          // Latch the control register data / dot data
          disp_ce_b <= 1'b1;
          dot_index <= 39;           // init for single char
          char_index <= 15;          // start with MS char
          state <= state+1;
          disp_rs <= 1'b0;           // Select the dot register
        end

      8'h06:
        begin
          // Load the user's dot data into the dot reg, char by char
          disp_ce_b <= 1'b0;
          disp_data_out <= dots[dot_index]; // dot data from msb
          if (dot_index == 0)
            if (char_index == 0)
              state <= 5;                    // all done, latch data
            else
            begin
              char_index <= char_index - 1;  // goto next char
              dot_index <= 39;
            end
          else
            dot_index <= dot_index-1;        // else loop thru all dots
        end

    endcase

always @ (data or char_index)
  case (char_index)
    4'h0:           nibble <= data[3:0];
    4'h1:           nibble <= data[7:4];
```

```
      4'h2:           nibble <= data[11:8];
      4'h3:           nibble <= data[15:12];
      4'h4:           nibble <= data[19:16];
      4'h5:           nibble <= data[23:20];
      4'h6:           nibble <= data[27:24];
      4'h7:           nibble <= data[31:28];
      4'h8:           nibble <= data[35:32];
      4'h9:           nibble <= data[39:36];
      4'hA:           nibble <= data[43:40];
      4'hB:           nibble <= data[47:44];
      4'hC:           nibble <= data[51:48];
      4'hD:           nibble <= data[55:52];
      4'hE:           nibble <= data[59:56];
      4'hF:           nibble <= data[63:60];
    endcase

  always @(nibble)
    case (nibble)
      4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
      4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
      4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
      4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
      4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
      4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
      4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
      4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
      4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
      4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
      4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
      4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
    endcase

endmodule
```

elijah_module.v

```
//DATA_STRUCT: {[P1,P2,K],[P1,P2,K].... n32}

//BOARD:
// n1  00  n2  00  n3  00  n4  00
// 00  n5  00  n6  00  n7  00  n8
// n9  00  n10 00  n11 00  n12 00
// 00  n13 00  n14 00  n15 00  n16
// n17 00  n18 00  n19 00  n20 00
// 00  n21 00  n22 00  n23 00  n24
// n25 00  n26 00  n27 00  n28 00
// 00  n29 00  n30 00  n31 00  n32


//MESSAGES:
//1.  Invalid move (moved to a piece occupied by opponent)
//2.  Invalid move (moved two pieces.)
//3.  Invalid move
//4.  You win
//5.  Opposition turn.
//6.  You lose.
//7.  Your turn.
//8.  Remove piece (press button when done)

/*
module retrieve_cv(input cv_enable, clock, input[3:0]buttons, output reg [31:0]cv_out, output reg receive_cv);
  parameter DEFAULT = 6'd0;
  parameter MOVE2 =6'd1;
  parameter MOVE3 =6'd2;
  parameter MOVE4 =6'd3;
  parameter MOVE5 =6'd4;
  parameter MOVE6 =6'd5;
  parameter MOVE7 =6'd6;
  parameter MOVE8 =6'd7;
  parameter MOVE8 =6'd7;

  reg receieve_cv_out;
  reg [31:0]cv;
  //loop over all board positions and create a player value and send to cv_out
  always @(posedge clock) begin
    if (cv_enable) begin

      case (buttons)
        DEFAULT: begin
          cv = 32'b00000000000000000000111111111111; //ALL PIECES IN ORIGINAL LOCATION
          receieve_cv_out = 1'b1;
        end

        MOVE2: begin
          cv = 32'b00000000000000000010110111111111; //n9 -> n13
          receieve_cv_out = 1'b1;
        end

        MOVE3: begin
          cv = 32'b00000000000001000000110111111111; //n13 -> n18
          receieve_cv_out = 1'b1;
        end
```

```
        MOVE4: begin
          cv = 32'b00000000010000000000110111111111; //n13 -> n22
          receieve_cv_out = 1'b1;
        end

        MOVE5: begin
          cv = 32'b00000000000000000001111011111111; //n8 -> n12
          receieve_cv_out = 1'b1;
        end

        MOVE6: begin
          cv = 32'b00000000000000000100101111111111;
          receieve_cv_out = 1'b1;
        end

        MOVE7: begin
          cv = 32'b00000000000000001000011111111111;
          receieve_cv_out = 1'b1;
        end

        MOVE8: begin
          cv = 32'b00000000000000000000111111111111;
          receieve_cv_out = 1'b1;
        end


      endcase


    end
    assign receive_cv = receieve_cv_out;
    assign cv_out = cv;
  end



endmodule

*/




module game_state(input clock, input[95:0]uart_game, input[63:0]cv_game, input reset, player, receive_cv,
receive_uart, end_turn_button, output reg transmit, cv_enable, output reg [95:0]transmit_game, output
reg[5:0]message_out, output reg[31:0] piece_removal, output reg[95:0] state_to_screen,
                                                 output reg[5:0]fsm, output reg[31:0] test_old, test_new, output
reg [7:0] number_removed, input resign_button);

parameter WAITING =6'd0;
parameter SAME_SPOT= 6'd1;
parameter ONE_PIECE= 6'd2;
parameter LAST_LINE =6'd3;
parameter SIMPLE_CHECK= 6'd4;
parameter JUMP_CHECK_1 =6'd5;
parameter JUMP_CHECK_2 =6'd6;
parameter JUMP_CHECK_3 =6'd7;
parameter KING =6'd8;
parameter CHECK_GAME_OVER =6'd9;
parameter INVALID =6'd10;
parameter SEND =6'd11;
```

```
parameter RECEIVE_WAITING =6'd12;
parameter GAME_OVER_RECEIVE =6'd13;
parameter REMOVE_PIECE =6'd14;
parameter REMOVAL_WAIT =6'd15;
parameter WAITING_CV =6'd16;
parameter END =6'd17;
parameter WAITING_SEND = 6'd18;
parameter REMOVAL_RECEIEVE = 6'd19;
parameter KING_JUMP_1 = 6'd20;
parameter KING_JUMP_2 = 6'd21;

  reg[5:0] state;
  reg[31:0] player1;
  reg[31:0] player2;
  reg[31:0] king;
  reg[95:0]previous_state;
  reg[95:0]current_state;

  reg[95:0]current_state_kingless;
  reg[95:0]previous_state_kingless;


  reg[95:0]diff_state;
  reg[95:0]old_position;
  reg[95:0]new_position;
  reg[95:0]middle_state;
  reg[31:0]middle_player;
  reg[31:0]diff_player;
  reg[7:0]old_position_int;
  reg[7:0]new_position_int;
  reg[7:0]previous_sum;
  reg[7:0]current_sum;
  reg[3:0]lastbits;
  reg[7:0]i;
  reg[63:0] august;

  reg[7:0]x_pos,y_pos;
        reg[7:0]game_over_count;
  reg[7:0]pieces_removed;

//outputs
  reg[31:0] piece_to_remove;
  reg[5:0] message;
  reg cv_enable_out;
  reg transmit_out;
  reg[95:0] transmit_game_out;


  always @(posedge clock) begin
   if (reset) begin
     piece_to_remove = 0;
     pieces_removed = 0'b0;
     if (player) begin
      //player1 logic
      current_state =
96'b010010010010010010010010010010010010000000000000000000000000100100100100100100100100100100100100; //default
board. IF YOU DONT CHANGE THIS, DANI OWES $2.. I changed it :/
      previous_state = 96'b0; //empty
      message = 5'd7;
      state = WAITING;
```

```
    end

     else begin
      current_state = 
96'b010010010010010010010010010010010010000000000000000000000000100100100100100100100100100100100100;
      previous_state = 96'b0;
      player1 = 32'b00000000000000000000111111111111;
      state = RECEIVE_WAITING;
      message = 5'd5;
     end
    end

    case (state)
      WAITING:begin

        if (end_turn_button) begin
          state = WAITING; //loop here until the send button is pressed.
        end
        else begin
          cv_enable_out = 1; //start the CV process.
          state = WAITING_CV;
        end
      end

      WAITING_CV: begin

            if (!receive_cv) begin
              state = WAITING_CV;
            end
            else begin
             cv_enable_out = 0;
             previous_state = current_state; //take current state and move to previous state.
              //remove the player 2 out of the current state.
              for (i=0; i<32; i= i+1) begin
                player2 = {current_state[1+3*i],player2[31:1]};
                king = {current_state[3*i],king[31:1]};
              end
              august = cv_game;
              //get cv values in.
              for (y_pos=0; y_pos<8; y_pos=y_pos+1) begin
                for (x_pos=0; x_pos<4; x_pos=x_pos+1) begin
                    player1[x_pos + y_pos*4] = august[(x_pos*2 + y_pos%2)*8 + y_pos];
                end
              end

              //generate current_state
              for ( i=0; i<32; i= i+1) begin
                current_state = {player1[i],player2[i],king[i],current_state[95:3]};
                //current_state_kingless = {player1[i],player2[i],1'b0,current_state[95:3]};
                                          //current_state[3*i] = current_state[3*i];
                                          //current_state[3*i+1] = current_state[3*i+1];
                                          //current_state[3*1+2] = player1[i];
              end

                                    current_state_kingless = current_state;
                                    for ( i=0; i<32; i= i+1) begin
                                      current_state_kingless[3*i] = 1'b0;
              end

              //MAKE VERSINO OF CURRENT STATE AND PREVIOUS STATE THAT SETS ALL KINGS TO 0.
```

```
              //create diff state and old position from this.
              //then when checking if its a king reference the actual current state.
              //

              previous_state_kingless = previous_state;
              for ( i=0; i<32; i= i+1) begin
                previous_state_kingless[i*3] = 1'b0;
              end




              diff_state = current_state_kingless ^ previous_state_kingless;   //XOR the two states together to
generate the change state values.

              old_position = previous_state_kingless & diff_state; //AND that with old and new to generate the two
state changes.
              new_position = current_state_kingless & diff_state;

                                   //diff_state = current_state ^ previous_state;
                                   //old_position = previous_state & diff_state;
                                   //new_position = current_state & diff_state;

                                    old_position_int = 0;
                                    new_position_int = 0;

              for ( i =0; i<32; i= i+1) begin
                old_position_int = old_position_int + old_position[2+i*3]*i;
                new_position_int = new_position_int + new_position[2+i*3]*i;

              end

              current_state[new_position_int*3] = previous_state[old_position_int*3];
                                      current_state[old_position_int*3] = 1'b0;

              state = SAME_SPOT;

            end

      end

      SAME_SPOT:begin
             //check to make sure no pieces are in same location
            if (player2[new_position_int] == 1) begin
              //declare that pieces are illegal and end the check.
              message = 5'd1;
              state = INVALID;
            end
            else begin
              state = ONE_PIECE;
            end
      end

      ONE_PIECE:begin

          previous_sum = 0;
                    current_sum = 0;
          for ( i=0; i<32; i= i+1) begin
              previous_sum = previous_sum + old_position[2+i*3];
              current_sum = current_sum + new_position[2+i*3];
```

```
          end

          if((previous_sum == 1) & (current_sum == 1)) begin
              state = LAST_LINE;
          end
          else begin
              message = 5'd2;
              state = INVALID;
       end

      end

      LAST_LINE:begin
          //if in previous state any pieces were on the last line then send to king state.

          state = SIMPLE_CHECK;

      end

      SIMPLE_CHECK: begin
          if (old_position_int == 0 | old_position_int == 7 | old_position_int == 8 | old_position_int == 15 |
old_position_int == 16 | old_position_int == 23 | old_position_int == 24) begin

              if (new_position_int == old_position_int + 4) begin
                  //valid position
                  state = SEND;
              end
              else begin
                  //invalid position
                  if (old_position_int >= 25) begin
                    state = KING;
                  end
                  else begin
                    state = JUMP_CHECK_1;
                  end
              end
          end
          else if (old_position_int == 4 | old_position_int ==5 | old_position_int ==6 | old_position_int == 12 |
old_position_int ==13 | old_position_int == 14 | old_position_int == 20 | old_position_int ==21 | old_position_int
==22 ) begin

              if (new_position_int == old_position_int +5 | new_position_int == old_position_int + 4) begin
                //valid position
                state = SEND;

              end
              else begin
                //invalid position
                 if (old_position_int >= 25) begin
                    state = KING;
                  end
                  else begin
                    state = JUMP_CHECK_1;
                  end
              end
          end
          else begin

              if (new_position_int == old_position_int +4 | new_position_int == old_position_int + 3) begin
                //valid position
```

```
            state = SEND;

          end
          else begin
            //invalid position
             if (old_position_int >= 25) begin
                state = KING;
              end
              else begin
                state = JUMP_CHECK_1;
              end
          end

      end

    end

    JUMP_CHECK_1: begin

      if ((old_position_int == 0) | (old_position_int == 8) | (old_position_int == 16)) begin

        if ((new_position_int == old_position_int + 9) && (player2[old_position_int + 4] == 1)) begin
          //remove opposition piece

          player2[old_position_int + 4] = 0;
          current_state[(old_position_int+4)*3+1] = 0;

          pieces_removed = pieces_removed + 1;

          state = CHECK_GAME_OVER;

        end
        else begin
          if (old_position_int >= 16) begin
            state = KING;
          end
          else begin
            state = JUMP_CHECK_2;
          end

        end

      end
                    else if ( (old_position_int == 7) | (old_position_int == 15) | (old_position_int == 23))
begin

        if ((new_position_int == old_position_int + 7) && (player2[old_position_int + 4] == 1)) begin
          //remove opposition piece

          player2[old_position_int + 4] = 0;
          current_state[(old_position_int+4)*3+1] = 0;

          pieces_removed = pieces_removed + 1;

          state = CHECK_GAME_OVER;

        end
        else begin
          if (old_position_int >= 16) begin
            state = KING;
```

```
              end
              else begin
                state = JUMP_CHECK_2;
              end

            end

          end

          else if (old_position_int == 4 | old_position_int ==5 | old_position_int ==6 | old_position_int == 12 |
old_position_int ==13 | old_position_int == 14 | old_position_int == 20 | old_position_int ==21 | old_position_int
==22 ) begin
             if ((new_position_int == old_position_int +9) && (player2[old_position_int+5]==1)) begin
            //jump right
              player2[old_position_int + 5] = 0;
              current_state[(old_position_int+5)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end

            else if  ((new_position_int == old_position_int + 7) && (player2[old_position_int + 4] == 1)) begin
            //jump left
              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end

            else begin
            //failed check.
              if (old_position_int >= 16) begin
                state = KING;
              end
              else begin
                state = JUMP_CHECK_2;
              end

            end

          end

          else begin

            if ((new_position_int == old_position_int +9) && (player2[old_position_int+4]==1)) begin
            //jump right
              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end
```

```
          else if  ((new_position_int == old_position_int + 7) && (player2[old_position_int + 3] == 1)) begin
          //jump left
            player2[old_position_int + 3] = 0;
            current_state[(old_position_int+3)*3+1] = 0;

            pieces_removed = pieces_removed + 1;

            state = CHECK_GAME_OVER;

          end

          else begin
          //failed check.
            if (old_position_int >= 16) begin
              state = KING;
            end
            else begin
              state = JUMP_CHECK_2;
            end

          end
        end
      end

      JUMP_CHECK_2: begin
        state = KING; //assume invalid and move on

        case (old_position_int)
          8'd0, 8'd8: begin

            if ((new_position_int == old_position_int +18)  && (player2[old_position_int+4]==1) &&
(player2[old_position_int+13]==1)) begin
              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              player2[old_position_int + 13] = 0;
              current_state[(old_position_int+13)*3+1] = 0;

              pieces_removed = pieces_removed + 2;

              state = CHECK_GAME_OVER;
            end
            else if ((new_position_int == old_position_int +16) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+12]==1)) begin
              player2[old_position_int + 4] = 0;
             current_state[(old_position_int+4)*3+1] = 0;

              player2[old_position_int + 12] = 0;
             current_state[(old_position_int+12)*3+1] = 0;

              pieces_removed = pieces_removed + 2;


             state = CHECK_GAME_OVER;

            end
            else begin
          //failed check.
              if (old_position_int >= 8) begin
                state = KING;
```

```
                end
                else begin
                  state = JUMP_CHECK_3;
              end
                                   end


            end
             8'd1, 8'd9: begin

              if ((new_position_int == old_position_int +18)  && (player2[old_position_int+4]==1) &&
(player2[old_position_int+13]==1)) begin
                player2[old_position_int + 4] = 0;
                current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 13] = 0;
                current_state[(old_position_int+13)*3+1] = 0;

                pieces_removed = pieces_removed + 2;


                state = CHECK_GAME_OVER;
              end
              else if ((new_position_int == old_position_int +16) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+12]==1)) begin
                player2[old_position_int + 4] = 0;
               current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 12] = 0;
               current_state[(old_position_int+12)*3+1] = 0;

                pieces_removed = pieces_removed + 2;

               state = CHECK_GAME_OVER;

              end

              else if ((new_position_int == old_position_int +16) && (player2[old_position_int+3]==1) &&
(player2[old_position_int+11]==1)) begin
                player2[old_position_int + 3] = 0;
                current_state[(old_position_int+3)*3+1] = 0;

                player2[old_position_int + 11] = 0;
               current_state[(old_position_int+11)*3+1] = 0;

               pieces_removed = pieces_removed + 2;


               state = CHECK_GAME_OVER;

              end

              else begin
            //failed check.
                if (old_position_int >= 8) begin
                  state = KING;
                end
                else begin
                  state = JUMP_CHECK_3;
              end
```

```
                                        end
            end
             8'd2, 8'd10: begin

              if ((new_position_int == old_position_int +14)  && (player2[old_position_int+3]==1) &&
(player2[old_position_int+10]==1)) begin
                player2[old_position_int + 3] = 0;
                current_state[(old_position_int+3)*3+1] = 0;

                player2[old_position_int + 10] = 0;
                current_state[(old_position_int+10)*3+1] = 0;
                pieces_removed = pieces_removed + 2;


                state = CHECK_GAME_OVER;
              end
              else if ((new_position_int == old_position_int +16) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+12]==1)) begin
                player2[old_position_int + 4] = 0;
               current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 12] = 0;
               current_state[(old_position_int+12)*3+1] = 0;

               pieces_removed = pieces_removed + 2;


               state = CHECK_GAME_OVER;

              end

              else if ((new_position_int == old_position_int +16) && (player2[old_position_int+3]==1) &&
(player2[old_position_int+11]==1)) begin
                player2[old_position_int + 3] = 0;
                current_state[(old_position_int+3)*3+1] = 0;

                player2[old_position_int + 11] = 0;
               current_state[(old_position_int+11)*3+1] = 0;

                pieces_removed = pieces_removed + 2;

               state = CHECK_GAME_OVER;

              end

              else begin
            //failed check.
                if (old_position_int >= 8) begin
                  state = KING;
                end
                else begin
                  state = JUMP_CHECK_3;
              end

                                        end
            end
             8'd3, 8'd11: begin

              if ((new_position_int == old_position_int +16)  && (player2[old_position_int+3]==1) &&
(player2[old_position_int+11]==1)) begin
```

```
                player2[old_position_int + 3] = 0;
                current_state[(old_position_int+3)*3+1] = 0;

                player2[old_position_int + 11] = 0;
                current_state[(old_position_int+11)*3+1] = 0;

               pieces_removed = pieces_removed + 2;

                state = CHECK_GAME_OVER;
              end
              else if ((new_position_int == old_position_int +14) && (player2[old_position_int+3]==1) &&
(player2[old_position_int+10]==1)) begin
                player2[old_position_int + 3] = 0;
               current_state[(old_position_int+3)*3+1] = 0;

                player2[old_position_int + 10] = 0;
               current_state[(old_position_int+10)*3+1] = 0;

              pieces_removed = pieces_removed + 2;

               state = CHECK_GAME_OVER;

              end
              else begin
            //failed check.
                if (old_position_int >= 8) begin
                  state = KING;
                end
                else begin
                  state = JUMP_CHECK_3;
              end
                                   end
            end
             8'd4, 8'd12: begin


              if ((new_position_int == old_position_int +18)  && (player2[old_position_int+5]==1) &&
(player2[old_position_int+14]==1)) begin
                player2[old_position_int + 5] = 0;
                current_state[(old_position_int+5)*3+1] = 0;

                player2[old_position_int + 14] = 0;
                current_state[(old_position_int+14)*3+1] = 0;

                pieces_removed = pieces_removed + 2;

                state = CHECK_GAME_OVER;
              end
              else if ((new_position_int == old_position_int +16) && (player2[old_position_int+5]==1) &&
(player2[old_position_int+13]==1)) begin
                player2[old_position_int + 5] = 0;
               current_state[(old_position_int+5)*3+1] = 0;

                player2[old_position_int + 13] = 0;
               current_state[(old_position_int+13)*3+1] = 0;

               pieces_removed = pieces_removed + 2;

               state = CHECK_GAME_OVER;
```

```
            end
            else begin
          //failed check.
              if (old_position_int >= 8) begin
                state = KING;
              end
              else begin
                state = JUMP_CHECK_3;
            end

          end
                                end

           8'd5, 8'd13: begin

           if ((new_position_int == old_position_int +18)  && (player2[old_position_int+5]==1) &&
(player2[old_position_int+14]==1)) begin
              player2[old_position_int + 5] = 0;
              current_state[(old_position_int+5)*3+1] = 0;

              player2[old_position_int + 14] = 0;
              current_state[(old_position_int+14)*3+1] = 0;

              pieces_removed = pieces_removed + 2;


              state = CHECK_GAME_OVER;
            end
            else if ((new_position_int == old_position_int +16) && (player2[old_position_int+5]==1) &&
(player2[old_position_int+13]==1)) begin
              player2[old_position_int + 5] = 0;
             current_state[(old_position_int+5)*3+1] = 0;

              player2[old_position_int + 13] = 0;
             current_state[(old_position_int+13)*3+1] = 0;

             pieces_removed = pieces_removed + 2;


             state = CHECK_GAME_OVER;

            end

            else if ((new_position_int == old_position_int +16) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+12]==1)) begin
              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              player2[old_position_int + 12] = 0;
             current_state[(old_position_int+12)*3+1] = 0;

             pieces_removed = pieces_removed + 2;


             state = CHECK_GAME_OVER;

            end

            else begin
          //failed check.
```

```
                if (old_position_int >= 8) begin
                  state = KING;
                end
                else begin
                  state = JUMP_CHECK_3;
              end
            end
                                end
             8'd6, 8'd14: begin

             if ((new_position_int == old_position_int +16)  && (player2[old_position_int+5]==1) &&
(player2[old_position_int+14]==1)) begin
                player2[old_position_int + 5] = 0;
                current_state[(old_position_int+5)*3+1] = 0;

                player2[old_position_int + 14] = 0;
                current_state[(old_position_int+14)*3+1] = 0;

                pieces_removed = pieces_removed + 2;


                state = CHECK_GAME_OVER;
              end
              else if ((new_position_int == old_position_int +16) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+12]==1)) begin
                player2[old_position_int + 4] = 0;
               current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 12] = 0;
               current_state[(old_position_int+12)*3+1] = 0;

               pieces_removed = pieces_removed + 2;


               state = CHECK_GAME_OVER;

              end

              else if ((new_position_int == old_position_int +14) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+11]==1)) begin
                player2[old_position_int + 4] = 0;
                current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 11] = 0;
               current_state[(old_position_int+11)*3+1] = 0;

               pieces_removed = pieces_removed + 2;


               state = CHECK_GAME_OVER;

              end

              else begin
            //failed check.
                if (old_position_int >= 8) begin
                  state = KING;
                end
                else begin
                  state = JUMP_CHECK_3;
```

```
              end
            end

                                    end
             8'd7, 8'd15: begin

              if ((new_position_int == old_position_int +16)  && (player2[old_position_int+4]==1) &&
(player2[old_position_int+12]==1)) begin
                player2[old_position_int + 4] = 0;
                current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 12] = 0;
                current_state[(old_position_int+12)*3+1] = 0;

               pieces_removed = pieces_removed + 2;

                state = CHECK_GAME_OVER;
              end
              else if ((new_position_int == old_position_int +14) && (player2[old_position_int+4]==1) &&
(player2[old_position_int+11]==1)) begin
                player2[old_position_int + 4] = 0;
               current_state[(old_position_int+4)*3+1] = 0;

                player2[old_position_int + 11] = 0;
               current_state[(old_position_int+11)*3+1] = 0;

              pieces_removed = pieces_removed + 2;

               state = CHECK_GAME_OVER;

              end
              else begin
            //failed check.
                if (old_position_int >= 8) begin
                  state = KING;
                end
                else begin
                  state = JUMP_CHECK_3;
              end


            end
                          end
          endcase
          //add logic to do 1 level of jump check
      end

      JUMP_CHECK_3: begin
          state = KING; //assume invalid and move on

          /*
          case (old_position_int)

          0'd0: begin

          end
          0'd1: begin

          end
          0'd2: begin
```

```
          end
          0'd3: begin

          end
          0'd4: begin

          end
          0'd5: begin

          end
          0'd6: begin

          end
          0'd7: begin

          end
          */

      end
      KING: begin
          //add logic to deal with king pieces.


          if (current_state[3*new_position_int]) begin
             if (old_position_int == 0 | old_position_int == 7 | old_position_int == 8 | old_position_int == 15 |
old_position_int == 16 | old_position_int == 23 | old_position_int == 24) begin

              if (new_position_int == old_position_int + 4 | new_position_int == old_position_int - 4 ) begin
                  //valid position
                  state = SEND;
              end
              else begin
                  //invalid position
                    state = KING_JUMP_1;

              end
          end
          else if (old_position_int == 4 | old_position_int ==5 | old_position_int ==6 | old_position_int == 12 |
old_position_int ==13 | old_position_int == 14 | old_position_int == 20 | old_position_int ==21 | old_position_int
==22 | old_position_int == 25 | old_position_int == 26 | old_position_int == 27) begin

              if (new_position_int == old_position_int +5 | new_position_int == old_position_int + 4 |
new_position_int == old_position_int -4 | new_position_int == old_position_int - 3) begin
                //valid position
                state = SEND;

              end
              else begin
                //invalid position

                    state = KING_JUMP_1;

              end
          end
          else begin

              if (new_position_int == old_position_int +4 | new_position_int == old_position_int + 3 |
new_position_int == old_position_int - 5 | new_position_int == old_position_int - 4) begin
                //valid position
```

```
                state = SEND;

              end
              else begin
                //invalid position

                    state = KING_JUMP_1;

              end

          end

        end

        else begin

          message = 5'd3;
         state = INVALID;

        end

      end

      KING_JUMP_1: begin
         //kings can't jump right now

          if ((old_position_int == 0) | (old_position_int == 8) | (old_position_int == 16) | (old_position_int
==24)) begin

            if ((new_position_int == old_position_int + 9) && (player2[old_position_int + 4] == 1)) begin
              //remove opposition piece

              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end

            else if ((new_position_int == old_position_int - 7) && (player2[old_position_int - 4] == 1)) begin
              //remove opposition piece

              player2[old_position_int - 4] = 0;
              current_state[(old_position_int-4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end

            else begin

                state = KING_JUMP_2;


            end
```

```
          end
       else if ( (old_position_int == 7) | (old_position_int == 15) | (old_position_int == 23) | (old_position_int
== 31)) begin

            if ((new_position_int == old_position_int + 7) && (player2[old_position_int + 4] == 1)) begin
              //remove opposition piece

              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end
            else if ((new_position_int == old_position_int - 9) && (player2[old_position_int - 4] == 1)) begin
              //remove opposition piece

              player2[old_position_int - 4] = 0;
              current_state[(old_position_int-4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end
            else begin

                state = KING_JUMP_2;


            end

          end

        else if (old_position_int == 4 | old_position_int ==5 | old_position_int ==6 | old_position_int == 12 |
old_position_int ==13 | old_position_int == 14
         | old_position_int == 20 | old_position_int ==21 | old_position_int ==22 | old_position_int == 29 |
old_position_int ==30 | old_position_int ==31  ) begin

             if ((new_position_int == old_position_int +9) && (player2[old_position_int+5]==1)) begin
            //jump right
              player2[old_position_int + 5] = 0;
              current_state[(old_position_int+5)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;

            end

            else if  ((new_position_int == old_position_int + 7) && (player2[old_position_int + 4] == 1)) begin
            //jump left
              player2[old_position_int + 4] = 0;
              current_state[(old_position_int+4)*3+1] = 0;

              pieces_removed = pieces_removed + 1;

              state = CHECK_GAME_OVER;
```

```
  end

  else if ((new_position_int == old_position_int -9) && (player2[old_position_int-4]==1)) begin
  //jump right
    player2[old_position_int -4] = 0;
    current_state[(old_position_int-4)*3+1] = 0;

    pieces_removed = pieces_removed + 1;

    state = CHECK_GAME_OVER;

  end

  else if  ((new_position_int == old_position_int - 7) && (player2[old_position_int -3] == 1)) begin
  //jump left
    player2[old_position_int -3] = 0;
    current_state[(old_position_int-3)*3+1] = 0;

    pieces_removed = pieces_removed + 1;

    state = CHECK_GAME_OVER;

  end

  else begin
  //failed check.

      state = KING_JUMP_2;


  end

end

else begin

  if ((new_position_int == old_position_int +9) && (player2[old_position_int+4]==1)) begin
  //jump right
    player2[old_position_int + 4] = 0;
    current_state[(old_position_int+4)*3+1] = 0;

    pieces_removed = pieces_removed + 1;

    state = CHECK_GAME_OVER;

  end

  else if  ((new_position_int == old_position_int + 7) && (player2[old_position_int + 3] == 1)) begin
  //jump left
    player2[old_position_int + 3] = 0;
    current_state[(old_position_int+3)*3+1] = 0;

    pieces_removed = pieces_removed + 1;

    state = CHECK_GAME_OVER;

  end

  else if ((new_position_int == old_position_int -9) && (player2[old_position_int-5]==1)) begin
  //jump right
```

```
          player2[old_position_int -5] = 0;
          current_state[(old_position_int-5)*3+1] = 0;

          pieces_removed = pieces_removed + 1;

          state = CHECK_GAME_OVER;

        end

        else if  ((new_position_int == old_position_int -7) && (player2[old_position_int -4] == 1)) begin
        //jump left
          player2[old_position_int -4] = 0;
          current_state[(old_position_int-4)*3+1] = 0;

          pieces_removed = pieces_removed + 1;

          state = CHECK_GAME_OVER;

        end

        else begin
        //failed check.

            state = KING_JUMP_2;


        end
      end

    end

    KING_JUMP_2: begin
      //kings can only jump once at the moment.
        message = 5'd3;
        state = INVALID;
    end


    CHECK_GAME_OVER: begin
        //system has just performed a jump and remvoed the
        if (player2 == 0) begin

          transmit_game_out = current_state; //send the current value outwards.
          transmit_out = 1'b1;
          message = 4;
          //send UART value

          state = END;
        end
        else begin
          state = SEND;

        end
    end

    INVALID: begin
                    if (resign_button) begin
          current_state =
96'b010010010010010010010010010010010010000000000000000000000000000000000000000000000000000000000000;
          transmit_game_out = current_state; //send the current value outwards.
```

```
      transmit_out = 1'b1;
      message = 6;
      state = END;

    end
    else begin

                              current_state = previous_state; //return to previous state
                              state = WAITING;
    end

end

SEND: begin

  if (new_position_int >= 28) begin
  //assign the piece as a king piece.
    current_state[new_position_int*3] = 1;
  end
  transmit_game_out = current_state; //send the current value outwards.
  transmit_out = 1'b1;
  message = 5;
  state = WAITING_SEND;

end

          WAITING_SEND: begin
                  transmit_out = 1'b0;
                  state = RECEIVE_WAITING;


          end

RECEIVE_WAITING: begin

  //loop here until a high on the receive is found.
  if (!receive_uart) begin
                    if(!end_turn_button) begin
                                  state = SEND;
                              end
                              else begin
                                 state = RECEIVE_WAITING;
                              end
  end
  else begin
    previous_state = current_state;
                    //current_state = uart_game;
    middle_state = uart_game; //pull in the current state.
                   //game_over_count = 0;
    for ( i=0; i<32; i= i+1) begin
      current_state[i*3] = middle_state[(31-i)*3];
      current_state[1+i*3] = middle_state[2+(31-i)*3];
      current_state[2+i*3] = middle_state[1+(31-i)*3];
      middle_player[i] = middle_state[1+(31-i)*3];
                              game_over_count = middle_state[2+(31-i)*3];
    end

           //if (game_over_count ==0) begin
                  //      message = 4;
                  //      state = END;
```

```
          //       end
          //       else begin
                          state = GAME_OVER_RECEIVE;
                  // end
  end
end

 GAME_OVER_RECEIVE: begin
  if (middle_player == 0) begin
      message = 6;
      state = END;
  end
  else begin
      state = REMOVE_PIECE;
  end
end

REMOVE_PIECE: begin
    //XOR the value middle_player and player1
    diff_player = middle_player ^ player1;
    if (diff_player == 0) begin
      message = 7;
      state = WAITING;
    end
    else begin
      message = 8;
      piece_to_remove = diff_player;
      state = REMOVAL_WAIT;
    end
end

REMOVAL_WAIT: begin

  if(end_turn_button) begin
    state = REMOVAL_WAIT;
  end
  else begin
    //run check to make sure that piece was actually removed.
     piece_to_remove = 0;
    cv_enable_out = 1; //start the CV process.
    state = REMOVAL_RECEIEVE;
    //TO COMPLETE

  end

end

REMOVAL_RECEIEVE: begin

   if (!receive_cv) begin
    state = WAITING_CV;
  end
  else begin
    cv_enable_out = 0;
    august = cv_game;
        //get cv values in.
    for (y_pos=0; y_pos<8; y_pos=y_pos+1) begin
      for (x_pos=0; x_pos<4; x_pos=x_pos+1) begin
          player1[x_pos + y_pos*4] = august[(x_pos*2 + y_pos%2)*8 + y_pos];
```

```
          end
        end

        state = REMOVE_PIECE;

      end

    end

    END: begin
        transmit_out = 1'b0;
        state = END;

    end

  endcase

  assign message_out = message;
  assign piece_removal = piece_to_remove;
  assign cv_enable = cv_enable_out;
  assign transmit = transmit_out;
  assign state_to_screen = current_state; //state for graphics
  assign fsm = state; //debug
  assign transmit_game = transmit_game_out;
  assign test_old = player1;
  assign test_new = {16'b0, old_position_int, new_position_int};
       //assign test_old = current_state_kingless[31:0];
       //assign test_new = current_state_kingless[63:32];

  assign number_removed = pieces_removed;

 end

endmodule

module state_machine_clock #(parameter DELAY=32'd10_000_000)(
    input clk,
    input reset,
    input sync,
    output out_sig
    );

  reg [31:0] counter;

  initial begin
    counter = DELAY;
  end

  assign out_sig = (counter == 0);

  always @(posedge clk) begin
    if (reset || sync)
      counter <= DELAY;
    else if (counter > 0)
      counter <= counter - 1;
    else
      counter <= DELAY;
  end
endmodule
```

graphics.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:23:07 11/18/2018
// Design Name:
// Module Name:    graphics
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module graphics(
   input [95:0] state,
   input [3:0] messages,
   input vclock,// 65MHz clock
   input [10:0] hcount,  // horizontal index of current pixel (0..1023)
   input [9:0]  vcount, // vertical index of current pixel (0..767)
   input hsync,         // XVGA horizontal sync signal (active low)
   input vsync,         // XVGA vertical sync signal (active low)
   input blank,         // XVGA blanking (1 means output black pixel)

   output reg phsync,   // pong game's horizontal sync
   output reg pvsync,   // pong game's vertical sync
   output reg pblank,   // pong game's blanking
   output [23:0] pixel  // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );

        reg bc_x_pol;
        reg bc_y_pol;
        reg [23:0] board_pixel;

        wire [7:0] border = 50;
        wire [10:0] width = 1023;
        wire [9:0] height = 767;
        wire [9:0] board_size = 512;
        wire [9:0] x_border = (width-board_size)/2;
        wire [9:0] y_border = (height-board_size)/2;

        reg [10:0] new_h;
        reg [9:0] new_v;

        reg [95:0] state_digest;
        reg [7:0] state_count_1;
        reg [7:0] state_count_2;
        reg [3:0] x_ind;
        reg [3:0] y_ind;
        reg [10:0] state_x;
        reg [9:0] state_y;
```

```
reg [10:0] delta_x;
reg [9:0] delta_y;
reg [20:0] big_sum;
reg [23:0] COLOR;

reg [3:0] which_layer = 4'h0;

always @(*) begin
        board_pixel = 24'h00_00_00;
        if ((hcount>x_border)&(hcount<(x_border+board_size))) begin
                if ((vcount>y_border)&(vcount<(y_border+board_size))) begin
                        //Define new coordinates within the game board
                        new_h = hcount-x_border;
                        new_v = vcount-y_border;

                        if (new_h % 128 > 64) begin
                                bc_y_pol = 1'b1;
                        end
                        else bc_y_pol = 1'b0;

                        if (new_v % 128 > 64) begin
                                bc_x_pol = 1'b1;
                        end

                        else bc_x_pol = 1'b0;

                        if (bc_x_pol^bc_y_pol) board_pixel = 24'h8B_45_13; //dark brown
                        else board_pixel = 24'hD2_B4_8C; //light brown

                        //Make pieces
                        state_digest = state;
                        state_count_1 = 0;
                        while (state_count_1 < 32) begin
                                //map the place in the state variable to a location on the grid
                                x_ind = 0;
                                y_ind = 0;
                                state_count_2 = 0;
                                while (state_count_2 < state_count_1) begin
                                        if (x_ind < 3) x_ind = x_ind + 1;
                                        else begin
                                                x_ind = 0;
                                                y_ind = y_ind + 1;
                                        end
                                        state_count_2 = state_count_2 + 1;
                                end

                                if (state_digest[2] == 1) COLOR = 24'hFF_00_00;
                                else if (state_digest[1] == 1) COLOR = 24'h00_FF_00;
                                else COLOR = board_pixel;

                                //find the center of the each piece on the board
                                state_y = y_ind*64 + y_border + 32;
                                if (y_ind%2 == 0) state_x = x_ind*128 + x_border + 32;
                                else state_x = x_ind*128 + x_border + 96;

                                delta_x = (hcount > state_x) ? (hcount - state_x) : (state_x - hcount);
                                delta_y = (vcount > state_y) ? (vcount - state_y) : (state_y - vcount);

                                big_sum = delta_y*delta_y + delta_x*delta_x;
```

```
                                if (big_sum <= 625) board_pixel = COLOR;

                                state_count_1 = state_count_1 + 1;
                                state_digest = state_digest>>3;
                        end
                end
        end
        phsync = hsync;
        pvsync = vsync;
        pblank = blank;
    end

    wire [23:0] your_turn_pixel;
    picture_blob #(.WIDTH(101),.HEIGHT(92))
        your_turn_blob(.x(10),.y(10),.hcount(hcount),.vcount(vcount),

    .message(messages),.pixel_clk(vclock),.pixel(your_turn_pixel));

    assign pixel = board_pixel + your_turn_pixel;

endmodule
```

graphics_2.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:23:07 11/18/2018
// Design Name:
// Module Name:    graphics
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module graphics2(
   input [95:0] state,
   input [5:0] messages,
        input [31:0] remove_piece,
        input [5:0] num_out,
   input vclock,// 65MHz clock
   input [10:0] hcount,  // horizontal index of current pixel (0..1023)
   input [9:0]  vcount, // vertical index of current pixel (0..767)
   input hsync,         // XVGA horizontal sync signal (active low)
   input vsync,         // XVGA vertical sync signal (active low)
   input blank,         // XVGA blanking (1 means output black pixel)
   input calibrate,

   output reg hsync2,   // pong game's horizontal sync
   output reg vsync2,   // pong game's vertical sync
   output reg blank2,   // pong game's blanking
   output [23:0] pixel  // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );

   reg [10:0] hcount1;
   reg [9:0] vcount1;
   reg hsync1;
   reg vsync1;
   reg blank1;

        reg [10:0] state_x1;
        reg [9:0] state_y1;

        reg bc_x_pol;
        reg bc_y_pol;
        reg [23:0] board_pixel;
        reg [23:0] board_pixel2;
        wire [23:0] your_turn_pixel;
        reg [23:0] your_turn_pixel2;
   wire [23:0] cal_pixel;
   wire [23:0] piece_pixel;
        reg [23:0] piece_pixel2;
```

```
	reg [5:0] piece_message = 9;

	wire [7:0] border = 50;
	wire [10:0] width = 1023;
	wire [9:0] height = 767;
	wire [9:0] board_size = 512;
	wire [9:0] x_border = (width-board_size)/2;
	wire [9:0] y_border = (height-board_size)/2;

	reg [10:0] new_h;
	reg [9:0] new_v;

	reg [10:0] x_ind;
	reg [9:0] y_ind;
	reg [11:0] index;
	reg [10:0] state_x;
	reg [9:0] state_y;
	reg [10:0] delta_x;
	reg [9:0] delta_y;
	reg [20:0] big_sum;
	reg [23:0] COLOR;

	reg [10:0] off_h;
	reg [9:0] off_v;

	reg [10:0] off_x_ind;
	reg [9:0] off_y_ind;
	reg [11:0] off_index;
	reg [10:0] off_state_x;
	reg [9:0] off_state_y;
	reg [10:0] off_delta_x;
	reg [9:0] off_delta_y;
	reg [20:0] off_big_sum;

	reg [3:0] which_layer = 4'h0;

reg [3:0] pipe_count = 1;

	parameter VSYNC_BLINK_COUNT = 15;
	reg [7:0] vsync_count = 0;
	reg remove_piece_blink = 0;
	always @(posedge vsync) begin
		vsync_count <= vsync_count + 1;
		if (vsync_count >= VSYNC_BLINK_COUNT) begin
			vsync_count <= 0;
			remove_piece_blink <= ~remove_piece_blink;
		end
	end

	always @(posedge vclock) begin
		hsync2 <= hsync1;
		hsync1 <= hsync;
		vsync2 <= vsync1;
		vsync1 <= vsync;
		blank2 <= blank1;
		blank1 <= blank;
		state_x1 <= state_x;
		state_y1 <= state_y;
		if (pipe_count > 1) begin
			hcount1 <= hcount;
```

```
                    vcount1 <= vcount;
                    your_turn_pixel2 <= your_turn_pixel;
                    piece_pixel2 <= piece_pixel;
                    pipe_count <= 1;
            end
    else if (pipe_count > 0) pipe_count <= pipe_count + 1;
            else pipe_count <= 1;
     end

     always @(*) begin
        board_pixel = 24'h00_00_00;
            if ((hcount>x_border)&(hcount<(x_border+board_size))) begin
                    if ((vcount>y_border)&(vcount<(y_border+board_size))) begin
                            //Define new coordinates within the game board
                            new_h = hcount-x_border;
                            new_v = vcount-y_border;

                            bc_y_pol = (new_h % 128 >= 64);
                            bc_x_pol = (new_v % 128 >= 64);

                            y_ind = new_v;
                            y_ind = y_ind>>6;
                            x_ind = new_h;
                            x_ind = x_ind>>7;

                            index = x_ind + 4*y_ind;

                            //show yellow background if that piece should be removed
                            //otherwise just do the regular checkerboard
                            if ((remove_piece[index] == 1) & (!(bc_x_pol^bc_y_pol)) & (index>0) &
(remove_piece_blink)) board_pixel = 24'hFF_FF_00;
                            else if (bc_x_pol^bc_y_pol) board_pixel = 24'h22_22_22; //dark grey
                            else board_pixel = 24'hCC_CC_CC; //light grey

                            //if (state[3*index + 2] == 1) COLOR = 24'hFF_00_00;
                            //if ((state[3*index] == 1)&(state[3*index + 1] == 1)) COLOR = 24'h00_FF_00;
                            //else if (state[3*index + 1] == 1) COLOR = 24'h00_00_FF;
                            //else COLOR = board_pixel;

                            //find the center of the each piece on the board
                            state_y = y_ind*64 + y_border + 32;
                            if (y_ind%2 == 0) state_x = x_ind*128 + x_border + 32;
                            else state_x = x_ind*128 + x_border + 96;

                            delta_x = (hcount > state_x1) ? (hcount - state_x1) : (state_x1 - hcount);
                            delta_y = (vcount > state_y1) ? (vcount - state_y1) : (state_y1 - vcount);

                            big_sum = delta_y*delta_y + delta_x*delta_x;

                            if ((state[3*index] == 1)&(state[3*index + 1] == 1)) begin
                                    COLOR = piece_pixel2;
                                    piece_message = 10;
                            end
                            else if (state[3*index + 1] == 1) begin
                                    COLOR = piece_pixel2;
                                    piece_message = 9;
                            end
                            else COLOR = board_pixel;

                            if (big_sum <= 625) board_pixel = COLOR;
```

```
				end
			end
			else if (hcount>(x_border+board_size)) begin
				//Show the pieces off-board
				off_h = hcount - (x_border + board_size);
				off_v = vcount - y_border;

				off_x_ind = off_h;
				off_x_ind = off_x_ind>>6;
				off_y_ind = off_v;
				off_y_ind = off_y_ind>>6;

				off_index = off_x_ind + 3*off_y_ind;

				if ((num_out > off_index) & (off_x_ind<3)) begin
					//find the center of the each piece on the board
					off_state_y = off_y_ind*64 + y_border + 32;
					off_state_x = off_x_ind*64 + board_size + x_border + 32;

					off_delta_x = (hcount > off_state_x) ? (hcount - off_state_x) : (off_state_x -
hcount);
					off_delta_y = (vcount > off_state_y) ? (vcount - off_state_y) : (off_state_y -
vcount);

					off_big_sum = off_delta_y*off_delta_y + off_delta_x*off_delta_x;

					if (off_big_sum <= 625) board_pixel = 24'h00_00_FF;
				end
			end
			board_pixel2 = board_pixel;
		end

		picture_blob #(.WIDTH(101),.HEIGHT(92))
			your_turn_blob(.x(10),.y(334),.hcount(hcount1),.vcount(vcount1),

		.message(messages),.pixel_clk(vclock),.pixel(your_turn_pixel));

   picture_blob #(.WIDTH(60),.HEIGHT(60))
			piece_blob(.x(state_x1-32),.y(state_y1-32),.hcount(hcount1),.vcount(vcount1),
				.message(piece_message),.pixel_clk(vclock),.pixel(piece_pixel));

   calibrate_display cal(.vclock(vclock),.hcount(hcount),.vcount(vcount),.width(width),
     .height(height),.board_size(board_size),.display(calibrate),.cd_pixel(cal_pixel));

		assign pixel = board_pixel2 + your_turn_pixel2 + cal_pixel;

endmodule
```

lab3.v

```
`default_nettype none
///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
///////////////////////////////////////////////////////////////////////////////

module debounce (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// 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)
//
// 2012-Sep-15: Converted to 24bit RGB
//
// 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 lab3   (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;

// 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 locked;
//assign clock_feedback_out = 0; // gph 2011-Nov-10

wire clk;
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(clock_65mhz), .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(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset));
assign reset = user_reset | power_on_reset;

     //Graphics Code
// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
           .hsync(hsync),.vsync(vsync),.blank(blank));

// 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);

// generate pixel value from reading ZBT memory
wire [7:0]   vr_pixel;
wire [17:0]  vr_pixel_clr;
wire [18:0]  vram_addr1;

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

// 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));

// decode color, sync, valid signals
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));

 // convert from YCrCB to RGB. 5 cycle delay`
 wire [23:0] RGB_conv;
 YCrCb2RGB ycrcb_2_rgb(.clk(tv_in_line_clock1), .rst(0),
```

```
                                                                  .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]),
.Cb(ycrcb[9:0]),
                                                                  .R(RGB_conv[23:16]), .G(RGB_conv[15:8]),
.B(RGB_conv[7:0]));

    // delay DV and FVH by 5 cycles to match RGB data before placing in RAM
    wire dv_delayed;
    wire [2:0] fvh_delayed;
    delayN #(.NDELAY(5)) __delay_dv   (.clk(tv_in_line_clock1), .in(dv), .out(dv_delayed));
    delayN #(.NDELAY(5)) __delay_fvh0 (.clk(tv_in_line_clock1), .in(fvh[0]), .out(fvh_delayed[0]));
    delayN #(.NDELAY(5)) __delay_fvh1 (.clk(tv_in_line_clock1), .in(fvh[1]), .out(fvh_delayed[1]));
    delayN #(.NDELAY(5)) __delay_fvh2 (.clk(tv_in_line_clock1), .in(fvh[2]), .out(fvh_delayed[2]));

    // code to write NTSC data to video memory

    wire [18:0] ntsc_addr;
    wire [35:0] ntsc_data;
    wire         ntsc_we;
        // Half-word stored at each ZBT addr. first 6 bits of R, G, B. Prev definition was ycrcb[29:22]
    // Use delayed DV and FVH to match RGB data
        wire [17:0] zbt_mem_in;
        assign zbt_mem_in = {RGB_conv[23:18], RGB_conv[15:10], RGB_conv[7:2]};
    ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh_delayed, dv_delayed, zbt_mem_in,
                    ntsc_addr, ntsc_data, ntsc_we, 1'b0);

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

    wire [18:0] vram_addr2 = count[0+18:0];
    wire [35:0] vpat = ( switch[1] ? {4{count[3+3:3],4'b0}}
                         : {4{count[3+4:4],4'b0}} );

    // mux selecting read/write to memory based on which write-enable is chosen

    wire        sw_ntsc = 1;//~switch[7];
    wire        my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
    wire [18:0] write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
    wire [35:0] write_data = sw_ntsc ? ntsc_data : vpat;

//   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;

    assign      vram_addr = my_we ? write_addr : vram_addr1;
    assign      vram_we = my_we;
    assign      vram_write_data = write_data;



   // select output pixel data
   // this is mostly garbage...
   reg [7:0]    pixel;
   wire [17:0] pixel_clr = vr_pixel_clr;
   reg  b,hs,vs;

   always @(posedge clk) begin
        pixel <= switch[0] ? {hcount[8:6],5'b0} : vr_pixel;
        b <= blank;
        hs <= hsync;
```

```
        vs <= vsync;
    end

        wire [7:0] hue_lo, hue_hi, sat_lo, val_lo;
        //assign dispdata[7:0] = hue_lo;
        //assign dispdata[15:8] = hue_hi;
        //assign dispdata[39:32] = sat_lo;
        //assign dispdata[47:40] = val_lo;
        hue_picker _hue_picker(.clk(clk),
                                                                          .b_up(button_up), .b_down(button_down),
                                                                          .b_left(button_left),
.b_right(button_right),
                                                                          .b_0(button0), .b_1(button1),
.b_2(button2), .b_3(button3),
                                                                          .hue_lo(hue_lo), .hue_hi(hue_hi),
.sat_lo(sat_lo), .val_lo(val_lo));

        wire [7:0] red   = {pixel_clr[17:12], 2'b0};
        wire [7:0] green = {pixel_clr[11:6],  2'b0};
        wire [7:0] blue  = {pixel_clr[5:0],   2'b0};

        wire [24:0] HSV_conv;
        wire [7:0] hue, saturation, value;
        rgb2hsv rgb_2_hsv(.clock(clk), .reset(0),
                                                         .r(red), .g(green), .b(blue),
                                                         .h(hue), .s(saturation), .v(value));

    // delay VGA signals by 22 to account for
        wire [10:0] hcount_hsv;
        wire [9:0]  vcount_hsv;
        wire hsync_hsv, vsync_hsv;
        vga_signal_delay #(.DELAY(22)) _vga_signal_delay(

        .clk(clk),

        .hcount_in(hcount), .vcount_in(vcount),

        .hsync_in(hsync), .vsync_in(vsync),

        .hcount_out(hcount_hsv), .vcount_out(vcount_hsv),

        .hsync_out(hsync_hsv), .vsync_out(vsync_hsv));
        // good for green: H 0x49 to 0x67, S 0x49, V 0xDD
    //   [THIS IS GOOD FOR RED]
    //wire mask = ~((hue > 8'h2C) & (hue < 8'hC7)) & (saturation > sat_lo) & (value > 50);
         wire mask = ~((hue>hue_lo)&(hue<hue_hi)) & (saturation>sat_lo) & (value>val_lo);
         //wire mask = (hue>hue_lo)&(hue<hue_hi) & (saturation>sat_lo) & (value>val_lo);
        //wire mask = (hue > 8'h2C) & (hue < 8'hC7) & (saturation > 8'h77) & (value > 50);
         wire mask_green_cal_marker = (hue>8'h49) & (hue<8'h67) & (saturation>8'h49) & (value>8'hDD);

         wire [7:0] mask2 = {8{mask}}; // expand bus


        wire [63:0] cv_game; //= 32'h0000_0000;
        wire transmit, cv_enable, receive;
        assign led[7:0] = ~cv_game[7:0];
        wire detector_indicator, cv_ready_out;
        wire [7:0] x_grid = {5'd0, switch[4:2]};
        wire [7:0] y_grid = {5'd0, switch[4:2]};
        wire [7:0] debug_counter;
```

```
        wire calib_good;
        detector _detector(.clk(clk), .vsync(vsync),
                                                                    .x_rgb(hcount), .y_rgb(vcount),
                                                                    .x_hsv(hcount_hsv), .y_hsv(vcount_hsv),
                                                                    .x_grid(x_grid), .y_grid(y_grid),
                                                                    .mask_pixel(mask),
                                                                    .mask_cal_pixel(mask_green_cal_marker),
                                                                    .detect_array_out(cv_game),
.indicator(detector_indicator),
                                                                    .debug_counter(debug_counter),
                                                                    .start_cv_in(cv_enable),
.ready_out(cv_ready_out),
                                                                    .calib_good(calib_good));
        // assign dispdata[55:48] = debug_counter;
   wire calibration_mode = switch[1]; // assign this to a switch or something
   wire [7:0] calmode_red   = detector_indicator ? 8'hFF : (switch[5] ? red : mask2);
   wire [7:0] calmode_green = detector_indicator ? 8'h00 : (switch[5] ? green : mask2);
   wire [7:0] calmode_blue  = detector_indicator ? 8'hFF : (switch[5] ? blue : mask2);

        assign user3[0] = vsync;
        assign user3[1] = cv_ready_out;

   // feed XVGA signals to user's pong game
   wire [23:0] rgb;
   wire [95:0] state; //= 96'h8A28_A28A_28A2_8A28_A28A_28A2;
   wire [3:0] test_message = switch[3:0];

   wire phsync,pvsync,pblank;
   /*graphics board_graphics(.vclock(clock_65mhz),.hcount(hcount),.state(state),.messages(test_message),
                .vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),
                .phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(rgb));*/
        wire [5:0] message_out, num_out;
        wire[31:0] piece_removal;
        wire hsync2;
        wire vsync2;
        wire blank2;
        graphics2 board_graphics_test(.vclock(clock_65mhz),.hcount(hcount),.state(state),.messages(message_out),

        .vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),.calibrate(switch[2]),.remove_piece(piece_remova
l),
                .num_out(num_out),.hsync2(hsync2),.vsync2(vsync2),.blank2(blank2),.pixel(rgb));

   // switch[1:0] selects which video generator to use:
   //  00: user's pong game
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
   //reg [23:0] rgb = pixel;

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red   = calibration_mode ? calmode_red   : rgb[23:16];
   assign vga_out_green = calibration_mode ? calmode_green : rgb[15:8];
   assign vga_out_blue  = calibration_mode ? calmode_blue  : rgb[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   //assign vga_out_blank_b = ~pblank;
        assign vga_out_blank_b = ~blank2;
   assign vga_out_pixel_clock = ~clock_65mhz;
   //assign vga_out_hsync = phsync;
   //assign vga_out_vsync = pvsync;
        assign vga_out_hsync = hsync2;
```

```
   assign vga_out_vsync = vsync2;

        //Game State Code
        wire game_clk;
        wire [95:0] uart_game; //= 96'h0000_0000_0000_0000_0000_0000;
        wire game_reset = switch[6];
        wire player = switch[7];
        wire[5:0]fsm_state;
        wire turn_button;
        debounce db_turn_button(.clock(clock_65mhz),.noisy(button1),.clean(turn_button));
        wire [95:0] transmit_game;

        wire[31:0] test_old, test_new;

        state_machine_clock game_clock(.clk(clock_65mhz),.reset(1'b0),.sync(1'b0),.out_sig(game_clk));
        game_state game(.clock(game_clk),.uart_game(uart_game),.cv_game(cv_game),.reset(game_reset),
                .player(player),.receive_cv(cv_ready_out),.receive_uart(receive),.end_turn_button(turn_button),
                .transmit(transmit),.cv_enable(cv_enable),.transmit_game(transmit_game),

        .message_out(message_out),.piece_removal(piece_removal),.state_to_screen(state),.fsm(fsm_state),.test_old(
test_old),.test_new(test_new),.number_removed(num_out),.resign_button(switch[3]));

        wire [63:0] data;
        //assign data = {test_old,test_new, 43'b0,fsm_state};
        assign data= {2'b0,fsm_state, 7'b0, calib_good, test_new[15:0], test_old};// [THIS IS REAL]
        //assign data[7:0] = hue_lo;
        //assign data[15:8] = hue_hi;
        //assign data[39:32] = sat_lo;
        //assign data[47:40] = val_lo;
        //assign data[59:56] = calib_good;
        display_16hex hexdisp1(reset, clock_27mhz, data,
                disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out);

        //UART code
        wire [95:0] received_state;
        wire [3:0] rx_state;
        wire [7:0] rx_count;
        naive_receive get_data(.rx(rs232_rxd),.clk(clock_65mhz),.state_out(uart_game),
                .public_state(rx_state),.public_rx_count(rx_count),.receive(receive));

   //assign led = rx_count;
        //assign led[0] = transmit;
        //assign led[1] = receive;
        //assign led[7:2] = 0;
        assign user4[5:2] = rx_state;
        assign user4[13:6] = rx_count;

        wire good_button;
        debounce dbb0(.clock(clock_65mhz),.noisy(button0),.clean(good_button));
        wire [95:0] accepted_state = {12{switch}};
        transmit send_data(.state_in(transmit_game),.transmit(transmit),
                .tx(rs232_txd),.clk(clock_65mhz));

        /*wire good_button;
        debounce dbb0(.clock(clock_65mhz),.noisy(button0),.clean(good_button));
        wire [95:0] accepted_state = {12{switch}};
        transmit send_data(.state_in(accepted_state),.transmit(good_button),
                .tx(user4[1]),.clk(clock_65mhz));*/ //This code will send at the push of button 0
```

```
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 [1:0]     hc4 = hcount[1:0];
   reg [17:0]     vr_pixel;
   reg [35:0]     vr_data_latched;
   reg [35:0]     last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc4[0]==0) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4[0]==1) ? vram_read_data : vr_data_latched;

   always @(*)           // each 36-bit word from RAM is decoded to 4 bytes
     case (hc4[0])
       0: vr_pixel = last_vr_data[17:0];
       1: vr_pixel = last_vr_data[35:18];
       //2'd1: vr_pixel = last_vr_data[7+16:0+16];
       //2'd0: vr_pixel = last_vr_data[7+24:0+24];
     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

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

        wire ram_clock;
   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 hue_picker(clk,
                                                b_up, b_down, b_left, b_right, b_0, b_1, b_2, b_3,
                                                hue_lo, hue_hi, sat_lo, val_lo);
        input clk;
        input b_up, b_down, b_left, b_right, b_0, b_1, b_2, b_3;
        output reg [7:0] hue_lo = 8'h1E;
        output reg [7:0] hue_hi = 8'hD6;
        output reg [7:0] sat_lo = 8'h65;
        output reg [7:0] val_lo = 8'd124;

        wire up, down, left, right, button0, button1, button2, button3;
        reg up_old, down_old, left_old, right_old,
                button0_old, button1_old, button2_old, button3_old;

        debounce _db_up   (.clock(clk), .reset(0), .noisy(b_up),    .clean(up));
        debounce _db_down (.clock(clk), .reset(0), .noisy(b_down),  .clean(down));
        debounce _db_left (.clock(clk), .reset(0), .noisy(b_left),  .clean(left));
        debounce _db_right(.clock(clk), .reset(0), .noisy(b_right), .clean(right));
        debounce _db_b_0  (.clock(clk), .reset(0), .noisy(b_0),     .clean(button0));
        debounce _db_b_1  (.clock(clk), .reset(0), .noisy(b_1),     .clean(button1));
        debounce _db_b_2  (.clock(clk), .reset(0), .noisy(b_2),     .clean(button2));
        debounce _db_b_3  (.clock(clk), .reset(0), .noisy(b_3),     .clean(button3));

        always @(posedge clk) begin
                if (up & ~up_old & hue_lo < 8'hFF)
                        hue_lo <= hue_lo + 1;
                if (down & ~down_old & hue_lo > 8'h00)
                        hue_lo <= hue_lo - 1;
                if (left & ~left_old & hue_hi > 8'h00)
                        hue_hi <= hue_hi - 1;
                if (right & ~right_old & hue_hi < 8'hFF)
                        hue_hi <= hue_hi + 1;

                if (button1 & ~button1_old & sat_lo > 8'h00)
```

```
                sat_lo <= sat_lo - 1;
        if (button0 & ~button0_old & sat_lo < 8'hFF)
                sat_lo <= sat_lo + 1;

        if (button2 & ~button2_old & val_lo < 8'hFF)
                val_lo <= val_lo + 1;
        if (button3 & ~button3_old & val_lo > 8'h00)
                val_lo <= val_lo - 1;

        up_old      <= up;
        down_old    <= down;
        left_old    <= left;
        right_old   <= right;
        button0_old <= button0;
        button1_old <= button1;
        button2_old <= button2;
        button3_old <= button3;
    end
endmodule
```

naive_receive.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/17/2018 06:11:20 PM
// Design Name:
// Module Name: receive
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module naive_receive(
    input rx,
    input clk,
    output reg [95:0] state_out,
    output [3:0] public_state,
    output [7:0] public_rx_count,
     output reg [17:0] check_yaself,
     output reg receive
    );

    reg [3:0] state = 4'b0000;
    reg [16:0] wait_count;
    reg still_low;
    reg [15:0] data_count;
    reg [8:0] tx_count;
     reg [126:0] big_rx;
    reg [108:0] raw_rx;
     reg [27:0] elijah_count = 0;
     reg [11:0] noise_count = 0;
     wire [17:0] CHECK = 18'b011011111111111101;

     reg [11:0] bit_sum;

    assign public_state = state;
    assign public_tx_count = tx_count;

     //assign receive = (state == 4'b0010);

    always @(posedge clk) begin
        if (state == 4'b0000) begin//waiting for a start bit
              //if (wait_count >= 500000) begin
              if (rx == 0) begin
                  state <= 4'b0001; //go to the next state if you detect a start bit
                  data_count <= 0;
                  tx_count <= 0;
```

```
                noise_count <= 0;
            end
            //end
            //else if (wait_count > 0) wait_count <= wait_count + 1;
            //else wait_count <= 0;
            //receive <= 0;
        end

          else if (state == 4'b0001) begin
                if (data_count == 500) begin
                    if (noise_count < 4) begin
                        state <= 4'b0011;
                        data_count <= 0;
                    end
                    else state <= 4'b0000;
                end
                else begin
                    if ((data_count % 16) == 0) noise_count <= noise_count + rx;
                    data_count <= data_count + 1;
                end
        end

        else if (state == 4'b0011) begin //accepting data
            if (tx_count == 127) begin
                    raw_rx <= big_rx[126:18];
                    check_yaself <= big_rx[17:0];
                    state <= 4'b0010;
                end
                if (data_count == 1000) begin
                     big_rx <= {rx, big_rx[126:1]};
                tx_count <= tx_count + 1;
                data_count <= 0;
            end
            else data_count <= data_count + 1;
        end

        else if (state == 4'b0010) begin //post-processing
                if (check_yaself == CHECK) begin
                    receive <= 1;
                    state_out <=
{raw_rx[105:90],raw_rx[87:72],raw_rx[69:54],raw_rx[51:36],raw_rx[33:18],raw_rx[15:0]};
                    //state_out <=
{big_rx[123:108],big_rx[105:90],big_rx[87:72],big_rx[69:54],big_rx[51:36],big_rx[33:18]};
                end
                else begin
                    receive <= 0;
                    state_out <= 96'hFF_00_00_00_00_FF;
                end
                if (elijah_count < 120000000) begin
                    elijah_count <= elijah_count + 1;
                    state <= 4'b0010;
                end
                else begin
                    elijah_count <= 0;
                    state <= 4'b0000;
                    receive <= 0;
                end
                wait_count <= 0;
            end
    end
```

```
endmodule
```

ntsc2zbt.v

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09  // gph mod 11/3/2011
//
//
// Bug due to memory management will be fixed. It happens because
// the memory addressing protocol is off between ntsc2zbt.v and
// vram_display.v. There are 2 solutions:
// -. Fix the memory addressing in this module (neat addressing protocol)
//    and do memory forecast in vram_display module.
// -. Do nothing in this module and do memory forecast in vram_display
//    module (different forecast count) while cutting off reading from
//    address(0,0,0).
//
// Bug in this module causes 4 pixel on the rightmost side of the camera
// to be stored in the address that belongs to the leftmost side of the
// screen.
//
// In this example, the second method is used. NOTICE will be provided
// on the crucial source of the bug.
//
/////////////////////////////////////////////////////////////////////////////
// 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'd30;
   parameter     ROW_START = 10'd30;

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

   reg [9:0]     col = 0;
   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 [19:0] x_delay;
   reg [19:0] y_delay;
   reg [1:0] we_delay;
   reg [1:0] eo_delay;

   always @ (posedge clk)
   begin
     x_delay <= {x_delay[9:0], x[1]};
     y_delay <= {y_delay[9:0], y[1]};
     we_delay <= {we_delay[1:0], we[1]};
     eo_delay <= {eo_delay[1:0], eo[1]};
   end

   // compute address to store data in
   wire [8:0] y_addr = y_delay[18:10];
        wire [9:0] x_addr = x_delay[19:10];

   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 [31:0] mydata2 = mydata;//{data[1],data[1],data[1],data[1]};
   wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[1], x_addr[7:0]};

   // 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[10]==0));

   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

// utility for creating coordinates from NTSC signals. Also indicates if the
// current data should be read and creates an edge detector on frame_start for
// each new frame
```

```
module ntsc_coordgen(vclk, fvh, dv, col, row, data_valid, frame_start);
        input vclk;
        input [2:0] fvh;
        input dv;
   output reg [9:0]       col = 0;
   output reg [9:0]       row = 0;
        output reg data_valid = 0;
        output frame_start;

        parameter        COL_START = 10'd30;
   parameter     ROW_START = 10'd30;

        reg old_dv;
        reg [2:0] old_fvh;
        assign frame_start = fvh[2] & ~old_fvh[2];

        always @(posedge vclk) begin
                old_dv <= dv;
                data_valid <= dv && !fvh[2] & ~old_dv;

                old_fvh <= fvh;

                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;
          end
        end
endmodule

module count_proximity(vclk, col, col_target, row, row_target, data_valid, frame_start, pixel, cnt);
        input vclk;
        input [9:0] col, col_target;
        input [9:0] row, row_target;
        input data_valid, frame_start;
        input pixel;
        output reg [7:0] cnt = 0;
        reg [7:0] internal_cnt = 0;

        parameter TARGET = 1; // desired pixel val
        parameter RADIUS = 5; // (square) search radius

        reg even_odd = 0;
        wire restart_ctr = (col == 10'd31 & row == 10'd31);
        always @(posedge vclk) begin
                if (data_valid) begin
                        even_odd <= restart_ctr ? ~even_odd : even_odd;
                        // reset at start of img
                        if (even_odd & restart_ctr) begin
                                cnt <= internal_cnt;
                                internal_cnt <= 8'h00;
                        end

                        // TODO edge conditions
                        if ( (col_target+RADIUS>col) & (col_target-RADIUS<col) &
                                                (row_target+RADIUS>row) & (row_target-RADIUS<row) )
                                internal_cnt <= internal_cnt + pixel;
                end
        end
```

```
endmodule

module centroid_calc(vclk, col, row, pixel, col_centroid, row_centroid);
        input vclk;
        input [9:0] col;
        input [9:0] row;
        input pixel;
        output [9:0] col_centroid;
        output [9:0] row_centroid;

        reg [32:0] x_sum = 0;
        reg [32:0] y_sum = 0;
        reg [32:0] x_count = 0;
        reg [32:0] y_count = 0;
        reg [32:0] x_coord = 0;
        reg [32:0] y_coord = 0;

        reg [32:0] _xrem, y_rem; // not used

        reg div_clk = 0;

        divider32(.clk(div_clk), .dividend(x_sum), .divisor(x_count), .quotient(x_coord), .remainder(_xrem));
        divider32(.clk(div_clk), .dividend(y_sum), .divisor(y_count), .quotient(y_coord), .remainder(_yrem));

        assign col_centroid = x_coord[9:0];
        assign row_centriod = y_coord[9:0];

        always @(posedge vclk) begin
                // start col/row ?
                if (col == 10'd30 & row == 10'd30) begin
                        div_clk <= 1;
                end
                else
                        div_clk <= 0;


        end

endmodule
```

picture_blob.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:50:00 11/27/2018
// Design Name:
// Module Name:    picture_blob
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//
// picture_blob: display a picture
//
//////////////////////////////////////////////////
module picture_blob
   #(parameter WIDTH = 101,     // default picture width
               HEIGHT = 86)    // default picture height
   (input pixel_clk,
    input [10:0] x,hcount,
    input [9:0] y,vcount,
    input [5:0] message,
    output reg [23:0] pixel);

   wire [20:0] image_addr;   // num of bits for a larger picture idk
   wire [7:0] image1_bits, image2_bits, image3_bits, image4_bits, image5_bits, image6_bits, image7_bits,
image8_bits;
   wire [7:0] red_mapped, green_mapped, blue_mapped, grey_mapped;
        reg [7:0] image_bits;

        wire [7:0] real_width = (message == 8) ? 86 : WIDTH;

   // calculate rom address and read the location
   //assign image_addr = (hcount-x) + (HEIGHT - (vcount-y)) * real_width;
        //let us rotate
        assign image_addr = (real_width-(vcount-y)) + (HEIGHT-(hcount-x))*real_width;

        //all the different roms
        your_turn rom1(.clka(pixel_clk), .addra(image_addr), .douta(image1_bits));
        illegal_move rom2(.clka(pixel_clk), .addra(image_addr), .douta(image2_bits));
        you_win_rom rom3(.clka(pixel_clk), .addra(image_addr), .douta(image3_bits));
        you_lose_rom rom4(.clka(pixel_clk), .addra(image_addr), .douta(image4_bits));
        remove_rom rom5(.clka(pixel_clk), .addra(image_addr), .douta(image5_bits));
        wait_rom rom6(.clka(pixel_clk), .addra(image_addr), .douta(image6_bits));

        its_fine_rom rom7(.clka(pixel_clk), .addra(image_addr), .douta(image7_bits));
        datboi_rom rom8(.clka(pixel_clk), .addra(image_addr), .douta(image8_bits));
```

```
   // use color map to create 8 bits R, 8 bits G, 8 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   // use color map to create 8bits R, 8bits G, 8 bits B;
   //red_coe rcm (.clka(pixel_clk), .addra(image_bits), .douta(red_mapped));
   //green_coe gcm (.clka(pixel_clk), .addra(image_bits), .douta(green_mapped));
   //blue_coe bcm (.clka(pixel_clk), .addra(image_bits), .douta(blue_mapped));

        //grey_color_map gcm (.clka(pixel_clk), .addra(image_bits), .douta(grey_mapped));

   // note the one clock cycle delay in pixel!
   always @(*) begin
     if ((hcount >= x && hcount < (x+HEIGHT)) &&
          (vcount >= y && vcount < (y+real_width))) begin
                if (message < 9) begin
                        if (message == 7) image_bits = image1_bits;
                        else if ((message == 1) | (message == 2) | (message == 3)) image_bits = image2_bits;
                        else if (message == 4) image_bits = image3_bits;
                        else if (message == 6) image_bits = image4_bits;
                        else if (message == 8) image_bits = image5_bits;
                        else if (message == 5) image_bits = image6_bits;
                        else image_bits = 0;
                        pixel = {image_bits, image_bits, image_bits}; // greyscale
                end
                else begin
                        if (message == 9) pixel = {8'h00, 8'h00, image7_bits};
                        else if (message == 10) pixel = {8'h00, image8_bits, 8'h00};
                        else pixel = 0;
                end
                end
          else pixel = 0;
   end
endmodule
```

piece.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    13:24:28 11/25/2018
// Design Name:
// Module Name:    piece
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////
//
// piece: generate circle on screen
//
//////////////////////////////////////////////////////////////////////
module piece
   #(parameter RADIUS = 11'h14,            // default width: 64 pixels
               COLOR = 24'hFF_FF_FF)  // default color: white
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [23:0] pixel);

        reg [10:0] delta_x;
        reg [9:0] delta_y;
        reg [20:0] big_sum;
        reg [21:0] RADIUS_SQ;

        always @(*) begin
                delta_x = (hcount > x) ? (hcount - x) : (x - hcount);
                delta_y = (vcount > y) ? (vcount - y) : (y - vcount);

                big_sum = delta_y*delta_y + delta_x*delta_x;
                RADIUS_SQ = RADIUS*RADIUS;

                if (big_sum <= RADIUS_SQ) pixel = COLOR;
                else pixel = 0;
        end

endmodule
```

receive.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:21:18 11/18/2018
// Design Name:
// Module Name:    comms
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module receive(
    output [95:0] state_out,
    output receive,
         input rx,                // serial data received by the FPGA
         input clk,
    );

        reg [3:0] state = 3'b0000;
   reg [10:0] sixteen_count;
   reg [3:0] ready_count;
   reg [16:0] wait_count;
   reg still_low;
   reg [15:0] data_count;
   reg [7:0] tx_count;

        reg [108:0] raw_rx;

        always @(posedge clk)
        begin

                //receiving
           if (state == 4'b0000) //waiting for a start bit
           begin
                        if (wait_count > 130000)
                        begin
                                if (tx == 0)
                                begin
                                        state <= 4'b0001; //go to the next state if you detect a start bit
                                        sixteen_count <= 0;
                                        ready_count <= 0;
                                end
                        end
                        else begin
                                if (tx == 0) wait_count <= 0; //reset if there is noise on the line. might be
too aggressive.
                                else wait_count <= wait_count + 1;
                        end
```

```
          end

          else if (state == 4'b0001) //verifying that the start bit was real
          begin
                    if (ready_count == 8) begin
                              state <= 4'b0011;
                              data_count <= 0;
                              tx_count <= 0;
                              raw_data <= {1'b0, raw_data[108:1]};
                    end
                    else begin
                              if (sixteen_count == 423)
                              begin
                                        if (tx == 1) begin
                                                    state <= 4'b0000; //go back to the waiting state i
                                                    wait_count <= 0;
                                        end
                                        else begin
                                                    ready_count <= ready_count + 1;
                                                    sixteen_count <= 0;
                                        end
                              end
                              else sixteen_count <= sixteen_count + 1;
                    end
          end

          else if (state == 4'b0011) //accepting data
          begin
                    if (tx_count == 109) state <= 4'b0010;
                    if (data_count == 6770) begin
                              raw_data <= {tx, raw_data[108:1]};
                              tx_count <= tx_count + 1;
                              data_count <= 0;
                    end
                    else data_count <= data_count + 1;
          end

          else if (state == 4'b0010) //post-processing
          begin
                    state_out <=
{raw_rx[105:90],raw_rx[87:72],raw_rx[69:54],raw_rx[51:36],raw_rx[33:18],raw_rx[15:0]};
                    state <= 4'b0000;
                    wait_count <= 0;
          end

        end

        assign tx = big_data_tx[0];
        assign receive = (state == 4'b0010);

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

endmodule
```

rgb2hsv.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:45:01 11/10/2010
// Design Name:
// Module Name:    rgb2hsv
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
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

                divider16 hue_div1(
                .clk(clock),
                .dividend(s_top),
                .divisor(s_bottom),
                .quotient(s_quotient),
                .fractional(s_remainder),
```

```
		.rfd(s_rfd)
		);
		divider16 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
```

transmit.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:21:18 11/18/2018
// Design Name:
// Module Name:    comms
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module transmit(
    input [95:0] state_in,
    input transmit,
         output tx,    // serial data sent to RS232 output driver
         input clk,
    output reg xmit_clk  // baud rate; sent to logic analyzer for debugging
    );

         // this section sets up the clk;
    wire [10:0] DIVISOR = 1000; // create 115,200 baud rate clock, not exact, but should work.
    reg [15:0] count;

    always @(posedge clk)
    begin
       count <= xmit_clk ? 0 : count+1;
       xmit_clk <= (count == DIVISOR-1);
    end

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

        reg [95:0] state_st = 96'hFF_FF_FF_FF_FF_FF;
        reg [126:0] big_data_tx;
        reg [11:0] t_count;
        reg [5:0] send_count;
        wire [15:0] CHECK_YOURSELF = 16'b1011111111111101;

        always @(posedge clk)
        begin
                //transmitting
           if (transmit == 1) begin
                        state_st <= state_in;
                        send_count <= 0;
                        t_count <= 0;
                end
                if (t_count >= 300) begin
                        if (send_count < 50) begin
                                big_data_tx <= {1'b1, state_st[95:80], 2'b01, state_st[79:64], 2'b01,
```

```
state_st[63:48], 2'b01, state_st[47:32], 2'b01, state_st[31:16], 2'b01, state_st[15:0], 2'b01, CHECK_YOURSELF,
2'b01};
                                send_count <= send_count + 1;
                        end
                        t_count <= 0;
                end
                else begin
                        if (count == 1) begin
                                big_data_tx <= {1'b1, big_data_tx[126:1]};
                                t_count <= t_count + 1;
                        end
                end
        end

        assign tx = big_data_tx[0];

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

endmodule
```

video_decoder.v

```
//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
//

/////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output       f;
   output       v;
   output       h;
   output       data_valid;
   // output [4:0] state;

   parameter    SYNC_1 = 0;
   parameter    SYNC_2 = 1;
   parameter    SYNC_3 = 2;
   parameter    SAV_f1_cb0 = 3;
   parameter    SAV_f1_y0 = 4;
   parameter    SAV_f1_cr1 = 5;
   parameter    SAV_f1_y1 = 6;
   parameter    EAV_f1 = 7;
   parameter    SAV_VBI_f1 = 8;
   parameter    EAV_VBI_f1 = 9;
   parameter    SAV_f2_cb0 = 10;
   parameter    SAV_f2_y0 = 11;
   parameter    SAV_f2_cr1 = 12;
   parameter    SAV_f2_y1 = 13;
   parameter    EAV_f2 = 14;
```

```
parameter     SAV_VBI_f2 = 15;
parameter     EAV_VBI_f2 = 16;




// In the start state, the module doesn't know where
// in the sequence of pixels, it is looking.

// Once we determine where to start, the FSM goes through a normal
// sequence of SAV process_YCrCb EAV... repeat

// The data stream looks as follows
// SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV sequence
// There are two things we need to do:
//   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
//   2. Decode the subsequent data

reg [4:0]     current_state = 5'h00;
reg [9:0]     y = 10'h000;  // luminance
reg [9:0]     cr = 10'h000; // chrominance
reg [9:0]     cb = 10'h000; // more chrominance

assign        state = current_state;

always @ (posedge clk)
  begin
     if (reset)
       begin

       end
     else
       begin
          // these states don't do much except allow us to know where we are in the stream.
          // whenever the synchronization code is seen, go back to the sync_state before
          // transitioning to the new state
          case (current_state)
            SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
            SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
            SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                                     (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                                     (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                                     (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
                                     (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                                     (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                                     (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                                     (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

            SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
            SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
            SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
            SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;

            SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
            SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
            SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
            SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

            // These states are here in the event that we want to cover these signals
            // in the future. For now, they just send the state machine back to SYNC_1
```

```
               EAV_f1: current_state <= SYNC_1;
               SAV_VBI_f1: current_state <= SYNC_1;
               EAV_VBI_f1: current_state <= SYNC_1;
               EAV_f2: current_state <= SYNC_1;
               SAV_VBI_f2: current_state <= SYNC_1;
               EAV_VBI_f2: current_state <= SYNC_1;

             endcase
          end
     end // always @ (posedge clk)

   // implement our decoding mechanism

   wire y_enable;
   wire cr_enable;
   wire cb_enable;

   // if y is coming in, enable the register
   // likewise for cr and cb
   assign y_enable = (current_state == SAV_f1_y0) ||
                     (current_state == SAV_f1_y1) ||
                     (current_state == SAV_f2_y0) ||
                     (current_state == SAV_f2_y1);
   assign cr_enable = (current_state == SAV_f1_cr1) ||
                      (current_state == SAV_f2_cr1);
   assign cb_enable = (current_state == SAV_f1_cb0) ||
                      (current_state == SAV_f2_cb0);

   // f, v, and h only go high when active
   assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;

   // data is valid when we have all three values: y, cr, cb
   assign data_valid = y_enable;
   assign ycrcb = {y,cr,cb};

   reg    f = 0;

   always @ (posedge clk)
     begin
        y <= y_enable ? tv_in_ycrcb : y;
        cr <= cr_enable ? tv_in_ycrcb : cr;
        cb <= cb_enable ? tv_in_ycrcb : cb;
        f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule



///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Register 0
```

```
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SELECT                             4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                               4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 1
///////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY                            2'h0
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                     1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT                       1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING                      1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                                  1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE                 1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM, `FOUR_TIMES_SAMPLING, `DIFFERENTIAL_INPUT,
`SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}

///////////////////////////////////////////////////////////////////////////////
// Register 2
///////////////////////////////////////////////////////////////////////////////

`define Y_PEAKING_FILTER                         3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
```

```
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                                  2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

///////////////////////////////////////////////////////////////////////////////
// Register 3
///////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                        2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                           4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                 1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                              1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT, `INTERFACE_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 4
///////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                       1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                              1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

///////////////////////////////////////////////////////////////////////////////
// Register 5
///////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                 4'b0000
```

```
`define GPO_0_1_ENABLE                          1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                          1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                     1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                            1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE, `GPO_0_1_ENABLE,
`GENERAL_PURPOSE_OUTPUTS}

///////////////////////////////////////////////////////////////////////////////
// Register 7
///////////////////////////////////////////////////////////////////////////////

`define FIFO_FLAG_MARGIN                        5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                              1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                    1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                     1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RESET, `FIFO_FLAG_MARGIN}

///////////////////////////////////////////////////////////////////////////////
// Register 8
///////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                         8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register 9
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                       8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST                       8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register B
```

```
////////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                              8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

////////////////////////////////////////////////////////////////////////////////
// Register C
////////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                    1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE          1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE                         6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE, `DEFAULT_VALUE_ENABLE}

////////////////////////////////////////////////////////////////////////////////
// Register D
////////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                        4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                        4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

////////////////////////////////////////////////////////////////////////////////
// Register E
////////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE              1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL             2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE                4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_CONTROL,
`TEMPORAL_DECIMATION_ENABLE}

////////////////////////////////////////////////////////////////////////////////
// Register F
////////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                      2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY              1'b0
```

```
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                    1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR                1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                         1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                        1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                              1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP, `POWER_DOWN_LLC_GENERATOR,
`POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY, `POWER_SAVE_CONTROL}

///////////////////////////////////////////////////////////////////////////////
// Register 33
///////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                       1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES               1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                             3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                              1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE, `AVERAGE_BIRIGHTNESS_LINES,
`PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
```

```
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB

`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
                    tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
```

```
   $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
   $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
   $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
   $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
   $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
end

//
// Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
//

reg [7:0] clk_div_count, reset_count;
reg clock_slow;
wire reset_slow;

initial
  begin
     clk_div_count <= 8'h00;
     // synthesis attribute init of clk_div_count is "00";
     clock_slow <= 1'b0;
     // synthesis attribute init of clock_slow is "0";
  end

always @(posedge clock_27mhz)
  if (clk_div_count == 26)
    begin
       clock_slow <= ~clock_slow;
       clk_div_count <= 0;
    end
  else
    clk_div_count <= clk_div_count+1;

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign reset_slow = reset_count != 0;

//
// I2C driver
//

reg load;
reg [7:0] data;
wire ack, idle;

i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
        .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
        .sda(tv_in_i2c_data));

//
// State machine
//

reg [7:0] state;
reg tv_in_reset_b;
reg old_source;
```

```
always @(posedge clock_slow)
   if (reset_slow)
     begin
        state <= 0;
        load <= 0;
        tv_in_reset_b <= 0;
        old_source <= 0;
     end
   else
     case (state)
       8'h00:
         begin
            // Assert reset
            load <= 1'b0;
            tv_in_reset_b <= 1'b0;
            if (!ack)
              state <= state+1;
         end
       8'h01:
         state <= state+1;
       8'h02:
         begin
            // Release reset
            tv_in_reset_b <= 1'b1;
            state <= state+1;
                 end
       8'h03:
         begin
            // Send ADV7185 address
            data <= 8'h8A;
            load <= 1'b1;
            if (ack)
              state <= state+1;
         end
       8'h04:
         begin
            // Send subaddress of first register
            data <= 8'h00;
            if (ack)
              state <= state+1;
         end
       8'h05:
         begin
            // Write to register 0
            data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
            if (ack)
              state <= state+1;
         end
       8'h06:
         begin
            // Write to register 1
            data <= `ADV7185_REGISTER_1;
            if (ack)
              state <= state+1;
         end
       8'h07:
         begin
            // Write to register 2
            data <= `ADV7185_REGISTER_2;
            if (ack)
```

```
        state <= state+1;
  end
8'h08:
  begin
     // Write to register 3
     data <= `ADV7185_REGISTER_3;
     if (ack)
       state <= state+1;
  end
8'h09:
  begin
     // Write to register 4
     data <= `ADV7185_REGISTER_4;
     if (ack)
       state <= state+1;
  end
8'h0A:
  begin
     // Write to register 5
     data <= `ADV7185_REGISTER_5;
     if (ack)
       state <= state+1;
  end
8'h0B:
  begin
     // Write to register 6
     data <= 8'h00; // Reserved register, write all zeros
     if (ack)
       state <= state+1;
  end
8'h0C:
  begin
     // Write to register 7
     data <= `ADV7185_REGISTER_7;
     if (ack)
       state <= state+1;
  end
8'h0D:
  begin
     // Write to register 8
     data <= `ADV7185_REGISTER_8;
     if (ack)
       state <= state+1;
  end
8'h0E:
  begin
     // Write to register 9
     data <= `ADV7185_REGISTER_9;
     if (ack)
       state <= state+1;
  end
8'h0F: begin
   // Write to register A
   data <= `ADV7185_REGISTER_A;
 if (ack)
   state <= state+1;
end
8'h10:
  begin
     // Write to register B
```

```
      data <= `ADV7185_REGISTER_B;
      if (ack)
        state <= state+1;
    end
  8'h11:
    begin
      // Write to register C
      data <= `ADV7185_REGISTER_C;
      if (ack)
        state <= state+1;
    end
  8'h12:
    begin
      // Write to register D
      data <= `ADV7185_REGISTER_D;
      if (ack)
        state <= state+1;
    end
  8'h13:
    begin
      // Write to register E
      data <= `ADV7185_REGISTER_E;
      if (ack)
        state <= state+1;
    end
  8'h14:
    begin
      // Write to register F
      data <= `ADV7185_REGISTER_F;
      if (ack)
        state <= state+1;
    end
  8'h15:
    begin
      // Wait for I2C transmitter to finish
      load <= 1'b0;
      if (idle)
        state <= state+1;
    end
  8'h16:
    begin
      // Write address
      data <= 8'h8A;
      load <= 1'b1;
      if (ack)
        state <= state+1;
    end
  8'h17:
    begin
      data <= 8'h33;
      if (ack)
        state <= state+1;
    end
  8'h18:
    begin
      data <= `ADV7185_REGISTER_33;
      if (ack)
        state <= state+1;
    end
  8'h19:
```

```
  begin
     load <= 1'b0;
     if (idle)
       state <= state+1;
  end

8'h1A: begin
   data <= 8'h8A;
   load <= 1'b1;
   if (ack)
     state <= state+1;
end
8'h1B:
  begin
     data <= 8'h33;
     if (ack)
       state <= state+1;
  end
8'h1C:
  begin
     load <= 1'b0;
     if (idle)
       state <= state+1;
  end
8'h1D:
  begin
     load <= 1'b1;
     data <= 8'h8B;
     if (ack)
       state <= state+1;
  end
8'h1E:
  begin
     data <= 8'hFF;
     if (ack)
       state <= state+1;
  end
8'h1F:
  begin
     load <= 1'b0;
     if (idle)
       state <= state+1;
  end
8'h20:
  begin
     // Idle
     if (old_source != source) state <= state+1;
     old_source <= source;
  end
8'h21: begin
   // Send ADV7185 address
   data <= 8'h8A;
   load <= 1'b1;
   if (ack) state <= state+1;
end
8'h22: begin
   // Send subaddress of register 0
   data <= 8'h00;
   if (ack) state <= state+1;
end
```

```
          8'h23: begin
             // Write to register 0
             data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
             if (ack) state <= state+1;
          end
          8'h24: begin
             // Wait for I2C transmitter to finish
             load <= 1'b0;
             if (idle) state <= 8'h20;
          end
       endcase

endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
          state <= 0;
          ack <= 0;
       end
     else
       case (state)
         8'h00: // idle
           begin
              scl <= 1'b1;
              sdai <= 1'b1;
              ack <= 1'b0;
              idle <= 1'b1;
              if (load)
                begin
                   ldata <= data;
                   ack <= 1'b1;
                   state <= state+1;
                end
           end
         8'h01: // Start
           begin
              ack <= 1'b0;
```

```
      idle <= 1'b0;
      sdai <= 1'b0;
      state <= state+1;
   end
 8'h02:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h03: // Send bit 7
   begin
      ack <= 1'b0;
      sdai <= ldata[7];
      state <= state+1;
   end
 8'h04:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h05:
   begin
      state <= state+1;
   end
 8'h06:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h07:
   begin
      sdai <= ldata[6];
      state <= state+1;
   end
 8'h08:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h09:
   begin
      state <= state+1;
   end
 8'h0A:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h0B:
   begin
      sdai <= ldata[5];
      state <= state+1;
   end
 8'h0C:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h0D:
   begin
```

```
      state <= state+1;
   end
 8'h0E:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h0F:
   begin
      sdai <= ldata[4];
      state <= state+1;
   end
 8'h10:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h11:
   begin
      state <= state+1;
   end
 8'h12:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h13:
   begin
      sdai <= ldata[3];
      state <= state+1;
   end
 8'h14:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h15:
   begin
      state <= state+1;
   end
 8'h16:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h17:
   begin
      sdai <= ldata[2];
      state <= state+1;
   end
 8'h18:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h19:
   begin
      state <= state+1;
   end
 8'h1A:
```

```
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1B:
  begin
     sdai <= ldata[1];
     state <= state+1;
  end
8'h1C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h1D:
  begin
     state <= state+1;
  end
8'h1E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1F:
  begin
     sdai <= ldata[0];
     state <= state+1;
  end
8'h20:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h21:
  begin
     state <= state+1;
  end
8'h22:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h23: // Acknowledge bit
  begin
     state <= state+1;
  end
8'h24:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h25:
  begin
     state <= state+1;
  end
8'h26:
  begin
     scl <= 1'b0;
     if (load)
       begin
```

```
                ldata <= data;
                ack <= 1'b1;
                state <= 3;
             end
           else
             state <= state+1;
         end
       8'h27:
         begin
            sdai <= 1'b0;
            state <= state+1;
         end
       8'h28:
         begin
            scl <= 1'b1;
            state <= state+1;
         end
       8'h29:
         begin
            sdai <= 1'b1;
            state <= 0;
         end
     endcase

endmodule
```

xvga.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:19:17 11/25/2018
// Design Name:
// Module Name:    xvga
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////////
module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current line
            output reg [9:0] vcount,      // line number
            output reg vsync,hsync,blank);

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   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
```

ycrcb2hsv.v

```
/***************************************************************************
 **
 ** Module: ycrcb2rgb
 **
 ** Generic Equations:
 ***************************************************************************/

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
```

zbt_6111.v

```
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

/////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                  ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                    // system clock
   input cen;                    // clock enable for gating ZBT cycles
   input we;                     // write enable (active HIGH)
   input [18:0] addr;            // memory address
   input [35:0] write_data;      // data to write
   output [35:0] read_data;      // data read from memory
   output        ram_clk;        // physical line to ram clock
   output        ram_we_b;       // physical line to ram we_b
   output [18:0] ram_address;    // physical line to ram address
   inout [35:0]  ram_data;       // physical line to ram data
   output        ram_cen_b;      // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire  ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};
```

```
   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = 1'b0;  // gph 2011-Nov-10
                                       // set to zero as place holder

//   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                    // times if its clk edges equal FPGA's
                                    // so we clock it on the falling edges
                                    // and thus let data stabilize longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111
```