Matthew Fox, Michael Kelessoglou, Evangelos Taratoris
6.111 Final Project Report
11 December 2013

# Virtual Pitch and Catch

## Project Overview

For our final project, we decided to make a highly interactive pitch and catch game. The game of pitch and catch resonates deeply within many cultures as a shared experience. Whether it be strengthening a parent-child relationship, passing time with a friend, or training for the upcoming season, people across the world engage in this game on a regular basis. On another strain, the tech community and the gaming industry as a whole are moving towards virtual reality as the next avenue for the new generation of games. One only has to look as far as the Nintendo Wii, XBox Kinect, Oculus Rift, or Playstation Move to see the trend of virtual reality sweeping the gaming industry.

Our idea for virtual pitch and catch lies at the intersection of these two cultural practices, the analog game of pitch and catch and the digital storm of virtual reality. We hope to bring the interactive game that has been so beloved for over a hundred years to the twenty-first century by creating a virtual interface for two players to throw and catch a digitally produced ball. We will use trackable gloves with multiple types of sensors to track and monitor the user's actions while playing the game, allowing us to show the game's progress on a monitor and simulate the actions of the virtual ball.

Furthermore, the implementation of the pitch and catch game provides a framework that is highly configurable for interesting applications. Some of these include a virtual basketball game, a three-dimensional pitch and catch game, a blue screen addition to the original game, and many more. This flexibility that the implementation allows supports the creation of this project that, in its basis, is already strong.

## Design Decisions

We divided the project into three main parts. These are the "Smart Glove" module, the Hand Tracking module, and the Physics and Display module. The project runs on two FPGA's, one of which is the master. The master is connected to one camera, one glove, and the VGA display, and runs the game logic. The other is connected to the other glove and another camera, and feeds the data from the Glove and Hand Tracking modules to the master. We opted to use two FPGA's because running hand-tracking from two different cameras on the same FPGA would be too memory intensive. The reason we decided to use two cameras rather than track both gloves with one, is that having one camera would severely restrict the distance at which the two players could be standing apart from each other. Figure 1 below shows a block

diagram, which indicates how these modules are interconnected. Though we did not consider it essential, we did also plan to include sound in the project, specifically a sound to indicate catching. At first we expected that we might need some sophisticated method of communication between the two FPGA's, perhaps even serial communication. However, we found out that the cameras and the gloves could easily be attached to really long wires, which allowed us to put the two FPGA's next to each other without restricting the distance between the players. Since the FPGA's don't run long wires between them, we found that simply sending the signals in parallel worked very well.

Another important design decision was making the game 2-dimensional to make the hand tracking, the physics, and especially the rendering simpler. If the game were 3D, the hand tracking would require two cameras for each glove, and would be very sensitive to their placement. The physics would not change much, as an extra component would just need to be added to all its position and velocity vectors. The display module would be completely different though, and would look less natural if implemented with sprites. If implemented without sprites, then gloves would have to somehow be represented by polygons, which would look too simplistic.

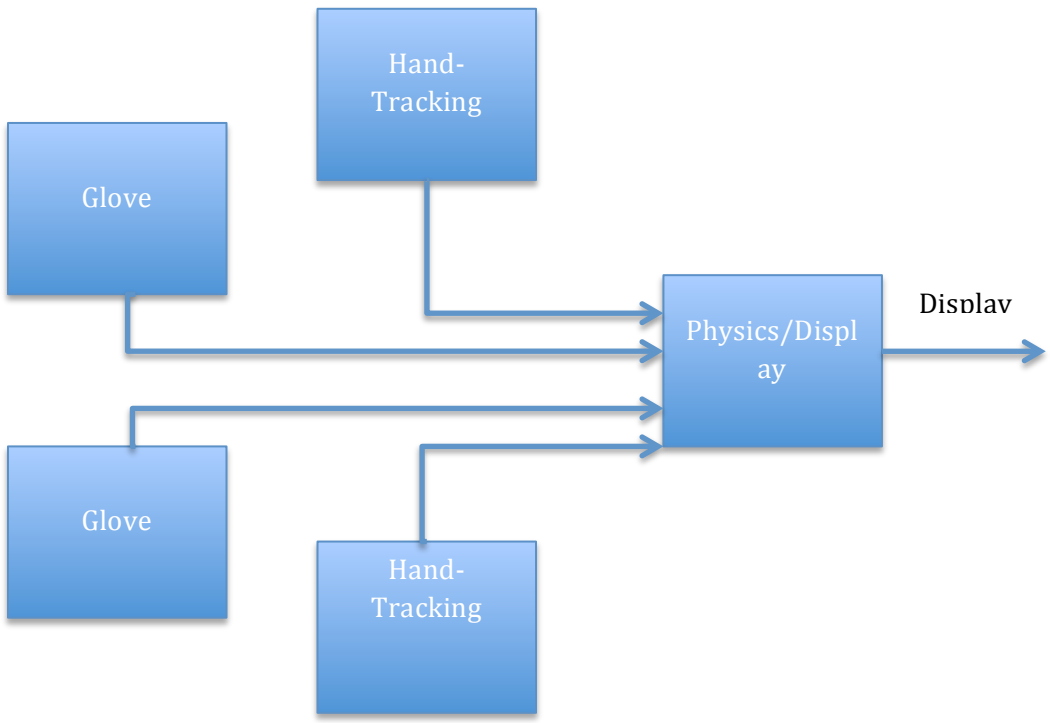


Figure 1: **Pitch and Catch Block Diagram** – This block diagram shows how two instances of the glove module and two instances of the hand-tracking module will interact with the Physics and Display module to simulate the game logic. The Physics/Display module will be able to render the game from the sensor data from the gloves and the tracking data from the cameras to show the progress of the game.

## Glove Module – Matthew Fox

Figure 2: The “Smart Glove” setup. Accelerometer mounted on the glove and flex sensors attached to the ring and index fingers.

The glove and its surrounding circuitry are one of the major modules of this project. We fitted two gloves that were intended to be tracked by the FPGA camera provided (one for each player). Because of this, we chose to buy a pair of bright orange gloves to serve as a bright indicator for the camera to track. These gloves were fitted with three sensors, namely two flex sensors and an accelerometer each. The flex sensors created a variable resistance that increases with bending. We fitted these to the outside of the index and ring fingers (see Figure 2 above) of the glove so as to discern when the hand closed. In the datasheet of these flex sensors, a suggested circuit is provided for producing an output voltage that will inform us of the level to which

the hand is closed. This circuit contained a voltage divider of which the flex sensor is half (it is the resistor connected to power). We paired the flex sensor with a 5.1 kΩ resistor as suggested in the datasheet. When the flex sensor bends, it gains resistance, so the voltage measured between the flex sensor and the other resistor drops from around 3.1 volts to 2.4 volts. After creating this circuit, we buffered the output to isolate the voltage divider from the following circuitry.

We then utilized an operational amplifier to push this output to the rails so that we could have a binary output from this module of the hand's openness. In order to do this, we chose 2.7 volts as the initial cutoff voltage to use in the operational amplifier circuit. By using a voltage divider (with a 4.7 kΩ and a 5.6 kΩ resistor), we generated the cutoff voltage and compared it to the voltage from our flex sensor circuit, creating our digital signal. By debouncing and synchronizing this signal with the system clock, we were able to complete this portion of the smart glove, having a reliable digital signal for the hand's state. Although we had chosen to have two flex sensors per hand to approach assuredness that the hand was intentionally being closed rather than partially flexed, we realized that taking the AND of the two signals would require absolute confidence that closing the hand would produce a low signal for both sensors every time. Because, in testing, this was proven to be incorrect, we decided instead to take the OR of these two values and modified the initial cutoff voltages to ensure that the circuits were not oversensitive. Each flex sensor was connected slightly differently because of the inexactness of sewing the sensors to the gloves, so we used slightly different voltage dividers for each sensor circuit to obtain the best signal possible. In this way, we created a flex sensor signal that both showed intention as well as ensuring near perfect accuracy with detecting a closing event. A picture of the wiring from the glove to the FPGA is shown in Figure 3 below.

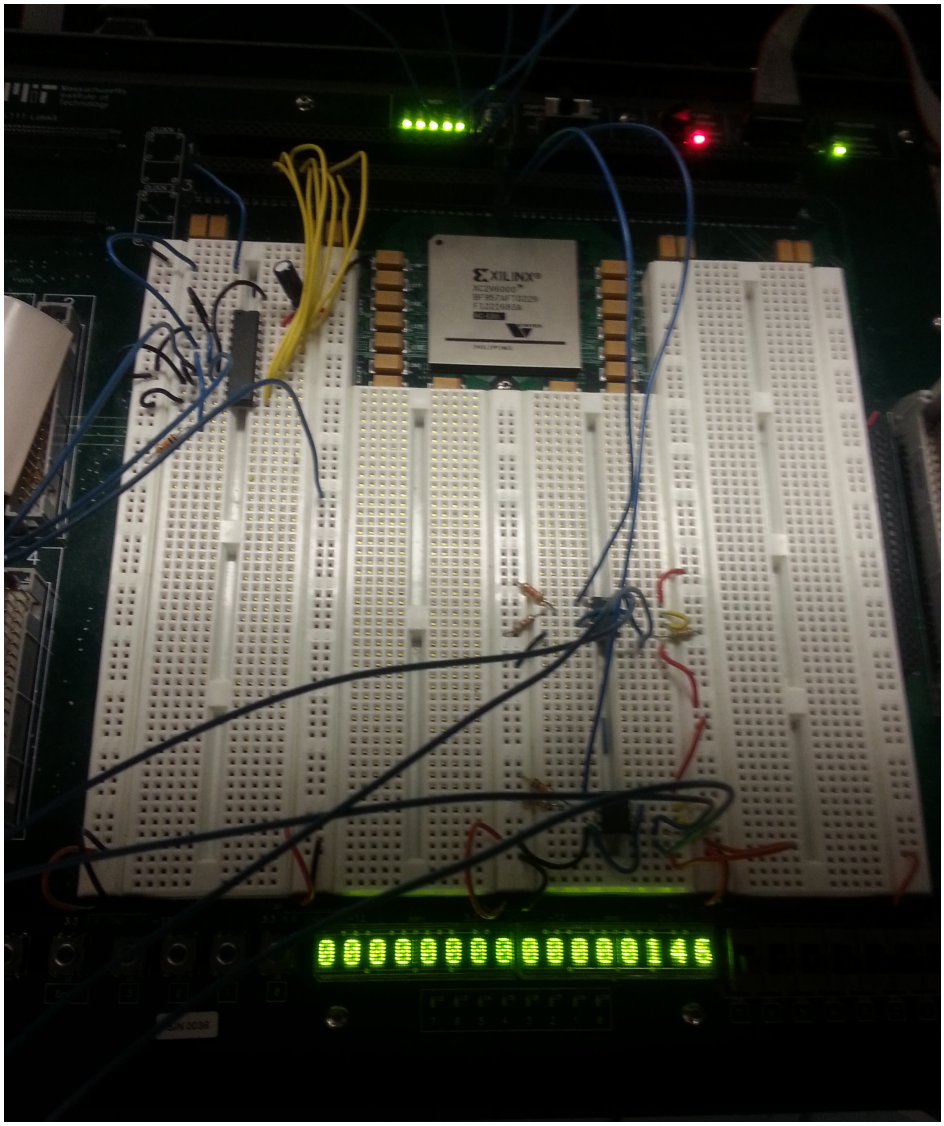

Figure 3: Wiring from “Smart Glove” to FPGA.

In addition to the bright glove and the flex sensors, we outfitted each glove with an accelerometer to capture the acceleration of the glove leading up to a throwing event. We ordered two accelerometers and breakout boards that were mounted on the back of the hand and intended to factor into the acceleration given to the ball in the physics module. We initially thought that the accelerometer should give reliable and near noise-free data that could counteract any issues that arise with noise in the hand-tracking module, allowing us to have multiple options should either the accelerometer or tracking fail to produce reliable results. However, soon after taking a look at the data outputted by the accelerometer, we realized that it is impossible to calculate position

just from the accelerometer. Because the accelerometer is but a force sensor, it is impossible to get a velocity measurement from just this data. In light of this new discovery, we repurposed the accelerometer to send a signal to the game engine about the orientation of the glove. We were able to discern whether the hand was palm up or down by setting a threshold value for the data from the z axis of the accelerometer. This method proved successful in telling if a player's hand was in a position where the ball could be caught or if the palm was facing away from the ball.

The act of getting the data from the accelerometer proved much more difficult than anticipated. Firstly, we made a design decision to place the two FPGAs close together to avoid the need to serialize our data that travels between the two (the canCatch signal, open signal, and glove position signal from the hand-tracking module). This decision was made possible by a long camera cord for the hand-tracking NTSC camera and necessitated a long cord from the glove sensors to the labkit. Unfortunately, the 3D accelerometer that we purchased originally was a digital accelerometer. Sending a digital signal across a long wire could compromise the integrity of the signal, so we opted to switch over to an analog accelerometer. The change to this analog accelerometer required an analog to digital conversion before we could input the data to the labkit. We decided to use an available ADC in lab that we put into a continuous conversion mode as per the datasheet to continuously update the data we received from the accelerometer for the z axis. This was one of the most difficult parts of this module as the circuit from the datasheet did not work as shown for some reason. By slightly changing the inputted clock data to synchronize the ADC to the labkit and adding a pull down resistor for the WR and INTR nodes, we were able to get correct functionality from the ADC and gain access to data for hand orientation.

Soldering the sensors onto the glove also proved to be much more difficult than we expected. Because of the length of the wire that we required, we opted to use a ribbon cable to neatly transfer all seven circuits together (2 circuits for each flex sensor and 3 circuits for the accelerometer – z data, power, and ground). This decision made soldering the sensors together much more difficult as the ribbon cable we acquired had multiple cores which forced us to use heat shrink tubing to isolate the circuits. Since the lab did not have heat shrink tubing or a heat gun, we borrowed supplies from the media lab and spent a large amount of time ensuring the complete connection of each circuit. After finally soldering these circuits to ensure strength and reliability, we had a working "Smart Glove" module that could be tested and integrated with the physics, display, and hand-tracking modules.

The glove module did not need inputs from the FPGA as it simply sent data back to the FPGA from the user's actions. The glove's outputs were an open signal and a canCatch signal. These inputs were synchronized between FPGAs before they could be used. Refer to Figure 4 below for the block diagram of the "Smart Glove" module.

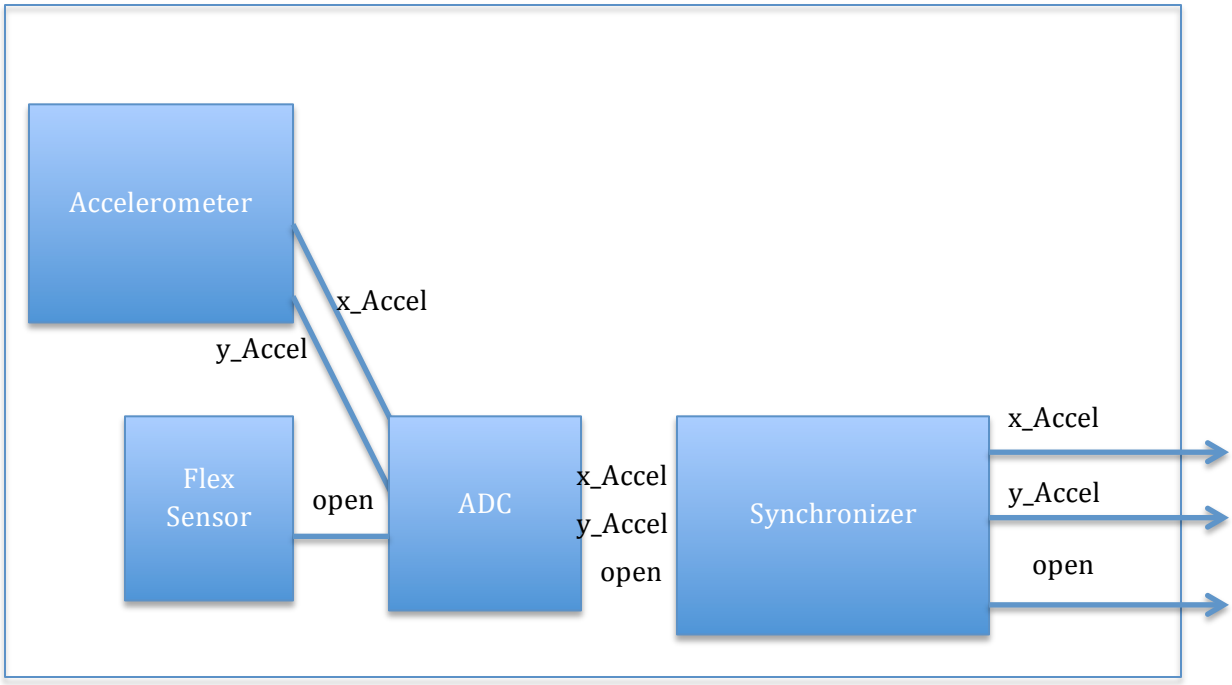


Figure 4: **The "Smart Glove" Module** – the "smart glove" will monitor its two flex sensors as well as its accelerometer. It will then convert these analog signals to digital signals and synchronize them with the system clock. These outputs will then go to the physics and display module.

If we were to start this module from the beginning, we would instead order an accelerometer and gyroscope pair to mount on the back of the hands. This would allow us to have enough data to calculate velocity and to have reliable tracking of the gloves regardless of the success of the hand-tracking module. We also think that the addition of the gyroscope would have made for better game play as we could have integrated sprites that move with your hand's exact orientation (just accelerometer data cannot exactly specify orientation). Unfortunately, we did not come to this idea until after the project had ended and we were evaluating ways to improve our performance. Nonetheless, the "Smart Glove" module proved to be a success and completed its purpose within the project as a whole.

## Hand-Tracking Module - Evangelos Taratoris

### Overview

The hand tracking module takes input from the camera. After locating the pixels that comprise the glove, it should calculate the center of mass and it should output a pair of values (X_COM, Y_COM) that correspond to the location of the center of mass.

**Implementation Details**

The input from the NTSC camera is in 10 bit Y, Cr, Cb colour format. Using the module *ycrcb2rgb* we translate those values into 8 bit red, green and blue values (R,G,B) in order to feed those signals to the screen display. We store two pixels worth of data in a single ZBT memory address. Since each location is 36 bits long, we have to store 18 bits of value for each pixel. Therefore we just drop the two least-significant-bits from each of the R,G and B signals.

When we read pixel data from the ZBT, it is helpful for our purposes, to translate the R,G and B values into H,S and V values (hue, saturation, value) using the module *rgb2hsv*. The H,S and V signals are 8 bits long and therefore range from 0-255 each. In general, this colour scheme performs well when we want to track a colour range, such as from red to orange. In the H,S,V space, after trying various ranges, we decided that an “acceptable” range for the orange colour should be:

14 <h<30, 235<s<255, 240<v<255

The *xvga* module outputs horizontal and vertical count signals (hcount and vcount). These are 11 bits and 10 bits long respectively. The visible range on our display is for 0<hcount<1024 and for 0<vcount<768. However, the camera output does not utilize this entire range. Instead only 30<hcount<750 and 30<vcount<510 are used to display the image from the camera.

**The *in_range* module**

This module takes as input the horizontal and vertical count (hcount,vcount) from the xvga module. In addition it takes as input the H,S and V values that we have calculated in the rgb2hsv module. It outputs a single bit named valid. Valid is 1 when the H,S and V values fall in the acceptable range **and** when the hcount and vcount fall in the range utilized by the camera. Otherwise valid is zero. This procedure in essence calculates whether we are going to accept a pixel as being within the colour range required to be considered orange and also within the hcount and vcount range to be considered in the camera display.

Following is the Verilog for this module:

```
module in_range(clk,h,s,v,hcount,vcount,valid);
```

```
reg valid;

always @(posedge clk)

valid <=
((hcount>11'd30)&&(hcount<11'd750)&&(vcount<10'd510)&&(vcount>10'd30)&&(h>8'd14)&&(h
<8'd30)&&(s>8'd235)&&(v>8'd240));

endmodule
```

### The hand_tracker module

This module takes as input hcount and vcount. In addition it takes as input the current valid value in addition to the last 7 valid values. We have saved those in a 8 bit register named valid 8. We update the values every clock cycle.

```
always @(posedge clk)
    if ((hcount==11'd30)&&(vcount>10'd30)&&(vcount<10'd510))
    begin
            valid8<=8'd0;
    end
    else
    begin
          valid8[7]<=valid8[6];
         …
          valid8[1]<=valid8[0];
          valid8[0]<=valid;
    end
```

The *hand_tracker* module outputs x_sum,y_sum and counter which are all 32 bit values. The module operates as follows:

At every clock cycle it checks whether the current valid bit as well as the previous 7 valid bits are all equal to 1. If this is the case, then it means that both the current pixel as well as the previous 7 pixels on the same line were all valid pixels. Hence it is highly likely that this pixel belongs to the glove. Therefore we increment two accumulators (x_accum, y_accum) with the value of the hcount and vcount of that pixel respectively. In addition we increment the value of a counter (counter_temp) to illustrate the fact that we encountered a pixel belonging to the glove. Every time that hcount and vcount are

equal to 30 (and therefore we begin a new camera frame) we set x_sum equal to x_accum, y_sum equal to y_accum and counter equal to counter_temp. We set the accumulators and the counter_temp to zero so that they can start again for the new frame. Therefore the outputs of this module change once per frame. We will use those outputs to calculate the center of mass of the glove.

Following is the Verilog for this module:

```
module hand_tracker(clk,valid8,hcount,vcount,x_sum,y_sum,counter);

always @(posedge clk)

begin

    if ((hcount==11'd30)&&(vcount==10'd30))

     begin

                x_sum<=x_accum;

                y_sum<=y_accum;

                  counter<=counter_temp;

                x_accum<=32'd0;

                   y_accum<=32'd0;

                  counter_temp<=32'd0;

      end

    else if (valid8==8'b11111111)

    begin

                   x_accum <= x_accum + {21'd0,hcount};

                   y_accum <= y_accum + {22'd0,vcount};

                   counter_temp <= counter_temp + 32'd1 ;

    end

end

endmodule
```

**Calculating the center of mass**

We use a 32-bit divider module which is provided by ISE to find the Center of Mass (COM_X,COM_Y) by dividing the accumulated values of x and y coordinates by the counter value.

Below is a simple implementation using Verilog:

```
wire [31:0] x_com,y_com,x_rem,y_rem;

wire ready_x,ready_y;

final_divider divider_x(.clk(clk),.dividend(x_sum_out),.divisor(counter_out),.quotient(x_com),
.fractional(x_rem),.rfd(ready_x));

 final_divider divider_y(.clk(clk),.dividend(y_sum_out),.divisor(counter_out),.quotient(y_com),
.fractional(y_rem),.rfd(ready_y));
```

**Low-Pass Filtering**

Even the most careful implementation of a colour-tracking algorithm may be problematic due to noise inherent in the video input. In order to tackle this, a simple low pass filter was created, where we just stored the previous 7 values of the calculated center of mass, and at each clock cycle, instead of returning the value of the recently calculated center of mass, we return the average of the previous 7 values and the current one. This does not require a divider since dividing by 8 is just a right shift of 3 bits.

```
assign x_com_out=(x_com+x_1+x_2+x_3+x_4+x_5+x_6+x_7)>>3;

assign y_com_out=(y_com+y_1+y_2+y_3+y_4+y_5+y_6+y_7)>>3;
```

**Testing/Debugging**

In order to check whether we were calculating the correct coordinates we utilized the *blob* module which outputs a square on screen whose upper left corner is specified as a pair of inputs (x,y). In our instance of the blob module we just set those values to be the x_com_out and y_com_out calculated by the low pass filter above.

Below is the code for the module and the instance we created.

```
module blob
  #(parameter WIDTH = 64,            // default width: 64 pixels
              HEIGHT = 64,           // default height: 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);

  always @ * begin
    if ((hcount >= x && hcount < (x+WIDTH)) &&
          (vcount >= y && vcount < (y+HEIGHT)))
         pixel = COLOR;
    else pixel = 0;
  end
endmodule

  blob #(.WIDTH(64),.HEIGHT(64),.COLOR(24'hFF_FF_00))   // yellow!
    paddle1(.x({x_com[10:2],2'b0}),.y({y_com[9:2],2'b0}),.hcount(hcount),.vcount(vcount),
          .pixel(blob_pixel));
```

The hand tracking module in its entirety did not function as intended. Most of the time there appeared to be tracking (i.e. the blob was close to the center of the glove). However the blob was flickering a lot, even when the glove was stable and sometimes it appeared really far away from the glove for brief instances of time. This problem can be due to the following 2 reasons or their combination:

i) The H, S, V range we defined as acceptable is too inclusive and therefore pixels that shouldn't be counted in the center of mass are actually taken into consideration.

ii) The filter mentioned above is not elaborate enough to prevent random noise affecting our center of mass calculation.

Even though various ranges were tested for the H,S,V inputs, we still didn't get a sufficiently good result. The mentioned filter was in essence the only one that we used due to time considerations. A member of the team is shown demonstrating the functionality of the tracking module in Figure 5 below.

Figure 5: The flickering of the square blob is obvious in this picture. In addition, the square would appear far away from the center of the glove for brief but visible periods.

**Implementation Process**

A considerable amount of time went into understanding how to get the video input from the camera, save it in the ZBT memory and then use it to perform the actual tracking.

In addition finding the correct range of H,S,V was a problem until the very end. A possible alternative would be to use R,G,B instead. Even though it appears that H,S,V is superior for colour tracking, using R,G,B ranges might end up giving us better tracking results.

The calculation of the center of mass uses dividers. Dividers in Verilog are really complicated objects and in general we try to avoid them. When we are generating the programming file, the dividers result in a long process that takes almost 15 minutes for

each programming file generation. However, we did not see a better way to perform this task that would not make use of the divider module.

In theory we could have made it so that we accept only a number of pixels that is a power of 2. Then, we could have easily calculated the center of mass of those pixels by using the shift operation, and the center of mass calculated would still be close to the actual center of mass. This is something that we would change, if we were to perform color tracking again.

## Physics Module – Michael Kelessoglou

### Overview

The Physics module uses the inputs from the Glove and Hand-Tracking modules to determine the game state. The game mechanics attempt to emulate real-world physics, with some exceptions to make the game more playable. One such exception is that there are invisible walls to which the ball sticks. This prevents the ball from flying sideways off the screen, which could potentially cause it to loop back and appear on the other side, which would be game-breaking.

### Implementation

The Physics module runs on a 65 MHz clock is divided into two submodules. The first converts the glove position data from the hand-tracking module to global coordinates. Global coordinates are unsigned and measured in millimeters to allow for good accuracy and make manipulations easy. The x coordinate is zero two meters to the left of the leftmost camera and the y coordinate is zero at ground level. Since hand tracking was not implemented, we used a dummy module, in which glove position is controlled by buttons on the FPGA. This module would have also taken the distance between the cameras and whether each player was playing right-handed to give the correct output, had we needed to implement it.

The second and much more complicated submodule of the Physics module is the Ball State Machine. This module has the global glove positions and the glove state from the Smart-Glove module as inputs and outputs the position and state of the ball and whether there is a catching event or not (for sound purposes). Besides the ball position and state, the module also keeps track of the ball's velocity, which allows it to keep track of its state. Ball velocity is kept in millimeters per second. Each velocity component is represented as an unsigned variable coupled with a direction bit, since that design is more robust than relying on signed arithmetic. The ball state is a two-bit

variable. When it is 0, it indicates that the ball is in the air. When it is 1, it indicates that the first glove is holding the ball, and when it is 2, it indicates that the second glove is holding the ball. The module uses a counter, which counts an appropriate number of cycles to set an update variable to 1 at a 128Hz rate. 128 was picked because it is a power of 2, which is convenient for calculation, and it is a high enough rate that it always updates in between frames on the VGA screen. If the ball state is 0, when the update signal is high, the y component of velocity is decreased by a constant (g*1s/128 in mm/s), and ball position is changed to its previous position plus velocity divided by 128. If the ball position is 1 or 2, then ball velocity is set to 128 times the difference between the current ball position and the ball position at the previous update. This is how initial velocity is set during a throw. Since ball position and velocity are unsigned variables coupled with direction bits, appropriate direction checks need to be made when manipulating these. When the ball is in state 0 and a glove closes (which is checked by passing it and its delayed inversion through an AND gate) and the glove is close enough to that ball, then the ball transitions to the state of being held by that glove (with glove1 having priority over glove2). When the ball is in states 1 or 2 and the glove holding it opens, it transitions to state 0.

## Display Module – Michael Kelessoglou

### Overview

The Display module gets the game state as an input from the Physics module and outputs the appropriate pixels to be displayed on the VGA screen. The game's appearance is rather minimalistic and retro, which fits its simple nature. A screenshot of the game and a close-up of one sprite are shown in Figures 6 and 7 below.

Figure 6: Screenshot of the game. Player 1 (left) is currently holding the ball, while Player 2 (right) has his hand closed. Ball seems yellow because of bad picture quality and is actually red.

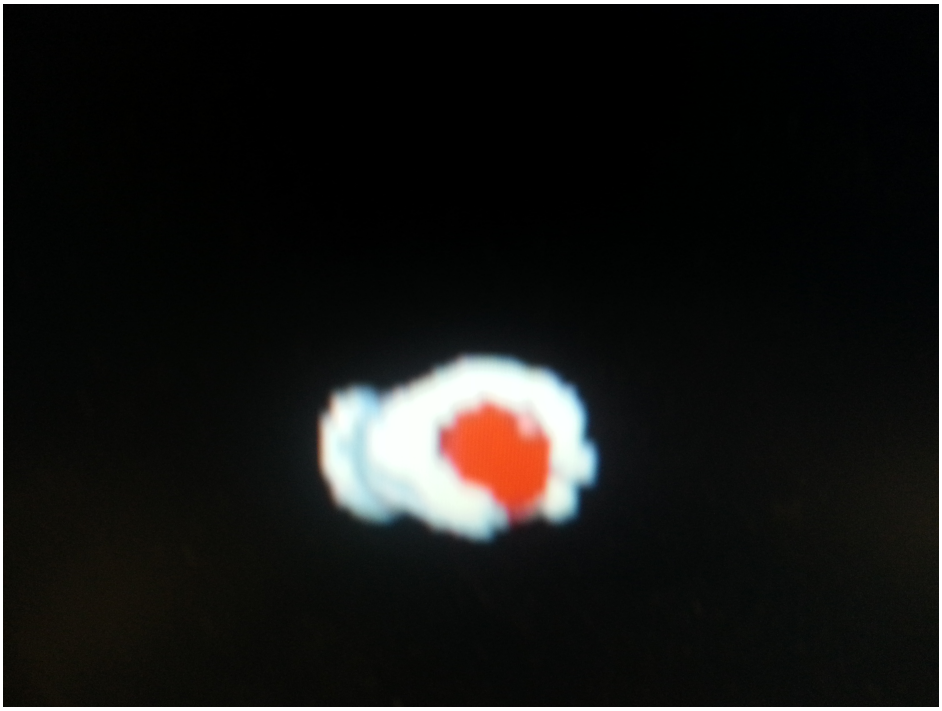

Figure 7: Close-up of the sprite of Player 1 (left hand glove) holding the ball.

**Implementation**

In order to display sprites, we had to first convert them to .coe files which are used to initialize ROMs on the FPGA. To make rendering easier, all sprites used had the same dimensions and the area in the sprite that was not part of the object was pure black. Since the FPGAs we used provide limited ROM space, we had to compress the sprite images first. To do this, each sprite was displayed using 16 colors, so that only 4 bits needed to be specified for each pixel. To convert bitmap images to .coe files, we ran a MATLAB script provided by the course staff, which also turned the 2-dimensional array of pixels into a 1-dimensional array.

The Display module runs on a 65 MHz clock in order to sync well with the VGA display. It is constituted of two submodules, one to convert the global coordinates of objects from the Physics module into pixel coordinates, and one to access the sprites from ROM and render them. The first submodule also takes in the distance between the two cameras as an input, so it knows how to scale the display. Specifically, the screen scales so that its 1024 pixel length corresponds to the lowest power of 2 in millimeters that is greater than the distance between the cameras plus 3 meters. The reason the scaling works this way is to avoid having to include a divider. Only having to divide by powers of 2 makes the module much simpler and faster, while the display scaling remains reasonable. The scaling happens by first calculating the base 2 logarithm of the length that will correspond to 1024 pixels and then right-shifting the coordinates by the log minus 10, since this is equivalent to multiplying by 1024 and dividing by the distance.

The second submodule of the Display module takes in hcount, vcount, the ball state, the glove states, and the pixel coordinates of an object and outputs the pixel corresponding to the object. It accomplishes this in 4 stages. The first stage checks whether the current hcount and vcount is outside the range and returns the address in the sprite's pixel map memory that the current pixel corresponds to. If the pixel is outside the range, it returns an address of 0, since the first pixel in all the sprites is black. The second stage is an instance of the sprite's pixel map ROM, which takes the address that is output from the first stage and returns the 4-bit entry in that address which corresponds to the address of the pixel's color in the sprite's color table. The third submodule is an instance of the sprite's color table ROM, which takes the address that is output from the second stage and outputs the 24-bit pixel entry. The first three stages are instantiated once for each sprite that could possibly represent the object. The fourth stage uses the ball state and glove states to determine which of the sprites is appropriate. For example, if a glove is closed but not holding the ball, the sprite where it is closed and not holding the ball will be chosen. The ball's output pixel will be black when it is being held, since it will then be shown in the glove's sprite.

The output pixels of each object are then combined in the following way. If the ball's pixel is not black, output the ball's pixel. Otherwise, if the left glove's pixel is not black, output the left glove's pixel. Otherwise, output the right glove's pixel. The check of whether it is black is very easy, as it is just a 24-bit NOR gate. The entire Display module is finely pipelined, which helps prevent display glitches, even during development, before we used the appropriate 65 MHz clock. The pipelining introduces 6

clock delays to the pixel output, so the hsync, vsync, and hblank signals are also delayed 6 times, so that the picture is not shifted. A block diagram of the physics and display modules is shown in Figure 8 below.

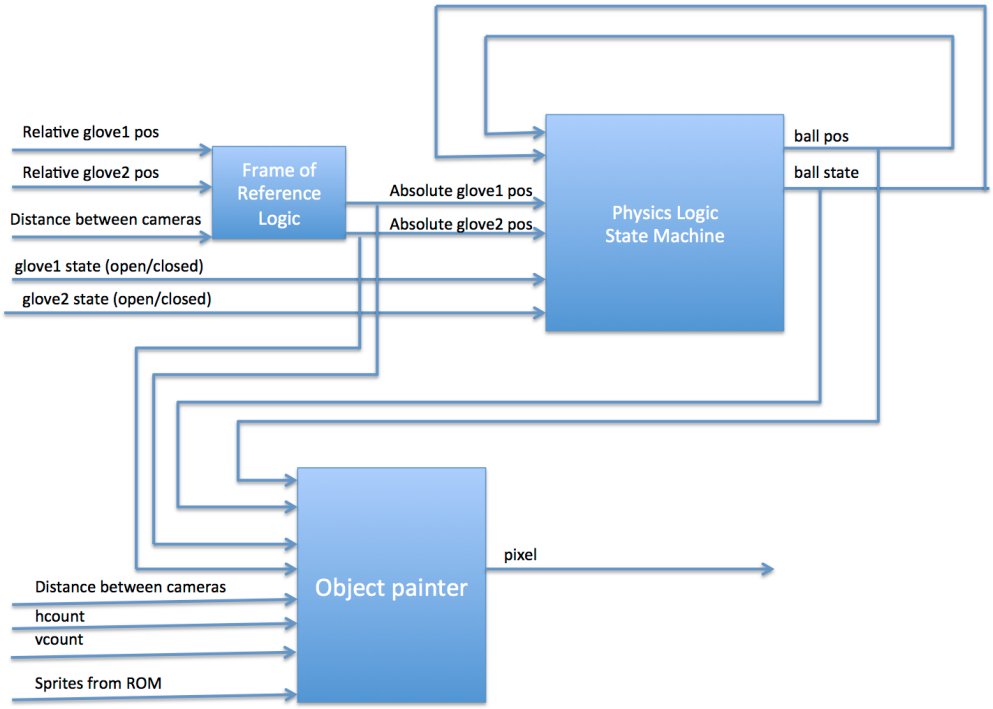


Figure 8: **Physics/Display Module** - The physics module is shown here with its inputs, outputs, and internal signals. This module will keep track of the ball, its motion, and its current state. The frame of reference will be computed in this module given the glove positions and input distance between the cameras. The ball will then be rendered by the object painter given its state from the physics state machine.

## Sound Module – Michael Kelessoglou, Matthew Fox

The Sound module gets a trigger signal from the ball state machine logic, which is high when the ball is caught. Our initial plan was to create a ROM from a .coe file with an appropriate sound, sampled at 44kHz (same as the AC97's sampling rate) and feed it to the AC97 sound chip to play it when triggered, but we could not get the right sound. Most of the Verilog we used was copied directly from lab 5 and then modified to fit our purpose. We debugged it to the point where we were sure that the problem was not in the .coe file or in the interface with the AC97 chip. Due to time constraints, we elected to use a pure tone generated by a mux with a sequentially incrementing set of select bits to cycle through all the values of the sine wave. We also applied a cycles limit, so that the tone would only last for a short period of time (about a second). Given more time we perhaps could have implemented our input sound in that fashion, but we're not sure that is feasible given the large amount of entries of a sound sampled at 44kHz.

## Conclusion

Though we were not able to make hand tracking reliable enough to interface with the other modules, we were able to successfully interface the Glove, Physics, and Display modules. The end result was the three modules listed above with button input providing glove position. This made for a playable and fun 2-player game, which was also easily expandable, and could be taken in various different directions, such as basketball or dodgeball. Given more time, we could have implemented tracking, which would have provided a much more immersive experience, as many modern game systems do, without losing its minimalistic and retro appeal.

Note: In the next few pages, we have provided the code for the combined Glove, Physics, and Display modules by file

## catch.v

```
module catch(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'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 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
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
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;
assign clock_feedback_out = 1'b0;
// 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[24] = 0;
    assign user4[29:0] = 30'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;
```

```
   ////////////////////////////////////////////////////////////////////////
   //
   // lab3 : a simple pong game
   //
   ////////////////////////////////////////////////////////////////////////

   // 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));

   // 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;

   // UP and DOWN buttons for pong paddle
   wire up,down;
   debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(up));
   debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(down));

   // 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));

   // feed XVGA signals to user's pong game
   wire [23:0] pixel;
   wire phsync,pvsync,pblank;

        //Matt's stuff////////////////////////////////////////////////////
        wire adcClk, canCatch, canCatchClean, openIndex, openRing, open;
```

```
  wire [7:0] xData;
  wire [7:0] yData;
  wire [7:0] zData;

  assign xData = user3[7:0];
  assign yData = user3[15:8];
  assign zData = user3[23:16];

  assign openIndex = user4[31];
  assign openRing = user4[30];

  assign open = openRing && openIndex;

  clk_divide div(.clk(clock_27mhz), .adcClk(adcClk));
  canCatch catcher(.clk(clock_27mhz), .zData(zData), .canCatch(canCatch));
  assign user3[31] = adcClk;

  //Debounce can Catch signal to make it more useful
  wire canCatchsync;
  debounce d0(.reset(0), .clock(clock_65mhz), .noisy(canCatchsync), .clean(canCatchClean));
  synchronize s0(.clk(clock_65mhz),.in(canCatch),.out(canCatchsync));

  wire [63:0] display;
  assign display = {3'b0, canCatch, 51'h0, open, zData};

  // Hex display setup
  display_16hex display1(.reset(0), .clock_27mhz(clock_27mhz), .data(display),
        .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs),
.disp_ce_b(disp_ce_b),
        .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));


        ////////////////////////////////////////////////////
        // signals for the game
        wire glove1closed;
        wire glove2closed;
        wire glove1x; // don't know format of coords yet
        wire glove1y;
        wire glove2x;
        wire glove2y;
        wire[5:0] dist;
        wire can_catch1;
        wire can_catch2;
        wire right_hand1;
        wire right_hand2;
        wire game_reset;
        wire soundtrigger;
        //will eventually be set by switches
```

```
      assign dist = 6'd6;
      assign can_catch1=canCatchClean;
      assign can_catch2=user1[1];
      assign glove1closed=~open;
      assign glove2closed=user1[0];
      assign game_reset = ~button_enter;
      ////////////////////////////////////////////////////


catch_game cg(.vclock(clock_65mhz),.reset(game_reset),
              .glove1closed(glove1closed),.glove2closed(glove2closed),
                                  .rel_glove1x(glove1x),.rel_glove1y(glove1y),
                                  .rel_glove2x(glove2x),.rel_glove2y(glove2y),
                                  .dist(dist),
                                  .test0(~button0),.test1(~button1),
                                  .testright(~button_right),.testleft(~button_left),
                                  .testup(~button_up),.testdown(~button_down),
                                  .testright2(user1[3]),.testleft2(user1[2]),
                                  .testup2(user1[4]),.testdown2(user1[5]),
                                  .can_catch1(can_catch1),.can_catch2(can_catch2),
                                  .right_hand1(right_hand1),.right_hand2(right_hand2),
              .hcount(hcount),.vcount(vcount),
                                 .sound(soundtrigger),
              .hsync(hsync),.vsync(vsync),.blank(blank),
                                  .debug(led[0]),
              .phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel));

// 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;
wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);

reg b,hs,vs;
always @(posedge clock_27mhz) begin
 hs <= phsync;
      vs <= pvsync;
      b <= pblank;
      rgb <= pixel;
end

// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = rgb[23:16];
assign vga_out_green = rgb[15:8];
assign vga_out_blue = rgb[7:0];
assign vga_out_sync_b = 1'b1;    // not used
```

```
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   assign led[7:1] = 7'b1111111;

        wire [4:0] volume = 5'b11111;

        reg synctrig;
        always @(posedge clock_27mhz) synctrig <= soundtrigger;

        sound s1(.clk(clock_27mhz),.trigger(synctrig),.ready(ready),.data(to_ac97_data));

        //assign led[1] = canCatch;
        // AC97 driver
   lab5audio a(clock_27mhz, reset, volume, to_ac97_data,ready,
               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
               ac97_synch, ac97_bit_clock);

endmodule

module canCatch(input clk, input [7:0] zData, output reg canCatch = 1);
   always @(posedge clk) begin
       if( zData > 8'h47 || zData == 8'h00 || zData == 8'h40) canCatch <= 1;
        else canCatch <= 0;
    end
endmodule

module clk_divide (input clk, output reg adcClk = 0);
  reg [7:0] counter;

  always @(posedge clk) begin
     if(counter == 16) begin
        adcClk <= !adcClk;
         counter <= 0;
     end
    else counter <= counter + 1;
  end
endmodule

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
```

## catch_game.v

```
module catch_game (
   input vclock,        // 27MHz clock
   input reset,               // 1 to initialize module
   input glove1closed,
        input glove2closed,
        input rel_glove1x,
        input rel_glove1y,
        input rel_glove2x,
        input rel_glove2y,
        input [5:0] dist,
        input can_catch1,
        input can_catch2,
        input right_hand1,
        input right_hand2,
        input test0,
        input test1,
        input testright,
        input testleft,
        input testup,
        input testdown,
        input testright2,
        input testleft2,
        input testup2,
        input testdown2,
   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 sound,
         output debug,
   output phsync,        // pong game's horizontal sync
   output pvsync,        // pong game's vertical sync
   output pblank,        // pong game's blanking
   output [23:0] pixel        // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );



        wire[15:0] glove1x;
        wire[15:0] glove1y;
        wire[15:0] glove2x;
        wire[15:0] glove2y;

        wire[1:0] ball_state;
        wire[15:0] ball_x;
```

```
wire[15:0] ball_y;

wire[10:0] ballpixelx;
wire[9:0] ballpixely;
wire[10:0] glove1pixelx;
wire[9:0] glove1pixely;
wire[10:0] glove2pixelx;
wire[9:0] glove2pixely;

wire[23:0] ballpixel;
wire[23:0] glove1pixel;
wire[23:0] glove2pixel;

wire catch_event;
wire throw_event;

assign sound = catch_event;

global_coords gc(.clk(vclock),
                .rel_glove1x(rel_glove1x),.rel_glove1y(rel_glove1y),
                .rel_glove2x(rel_glove2x),.rel_glove2y(rel_glove2y),
                .dist(dist),.right_hand1(right_hand1),.right_hand2(right_hand2),
                .testright(testright),.testleft(testleft),
                .testup(testup),.testdown(testdown),
                .testright2(testright2),.testleft2(testleft2),
                .testup2(testup2),.testdown2(testdown2),
                .glob_glove1x(glove1x),.glob_glove1y(glove1y),
                .glob_glove2x(glove2x),.glob_glove2y(glove2y));

ballSM bsm(.clk(vclock),.reset(reset),
                .glove1x(glove1x),.glove1y(glove1y),
                .glove2x(glove2x),.glove2y(glove2y),
                .glove1closed(glove1closed),.glove2closed(glove2closed),
                .can_catch1(can_catch1),.can_catch2(can_catch2),
                .test0(test0),.test1(test1),
                .dist(dist),
                .debug(debug),
                .catch_event(catch_event),.throw_event(throw_event),
                .ball_state(ball_state),.ball_x(ball_x),.ball_y(ball_y));


coords_to_pixel bpc(.x_coord(ball_x),.y_coord(ball_y),
                .dist(dist),
                .pixel_x(ballpixelx),.pixel_y(ballpixely));



coords_to_pixel g1pc(.x_coord(glove1x),.y_coord(glove1y),
```

```
                .dist(dist),
                .pixel_x(glove1pixelx),.pixel_y(glove1pixely));

        coords_to_pixel g2pc(.x_coord(glove2x),.y_coord(glove2y),
                .dist(dist),
                .pixel_x(glove2pixelx),.pixel_y(glove2pixely));


/*        blob bblob(.x(ballpixelx),.hcount(hcount),
                .y(ballpixely),.vcount(vcount),
                .color({8'hFF,16'b0}),
                .pixel(ballpixel));*/

        draw_ball db(.clk(vclock),.hcount(hcount),.vcount(vcount),.blank(blank),
                .x(ballpixelx),.y(ballpixely),.ball_state(ball_state),
                .pixel(ballpixel));



/*        blob g1blob(.x(glove1pixelx),.hcount(hcount),
                .y(glove1pixely),.vcount(vcount),
                .color({16'b0,8'hFF}),
                .pixel(glove1pixel));*/

        draw_p1 dp1(.clk(vclock),.hcount(hcount),.vcount(vcount),.blank(blank),
                .x(glove1pixelx),.y(glove1pixely),.ball_state(ball_state),
                .closed(glove1closed),.pixel(glove1pixel));

        /*blob g2blob(.x(glove2pixelx),.hcount(hcount),
                .y(glove2pixely),.vcount(vcount),
                .color({16'b0,8'hFF}),
                .pixel(glove2pixel));*/

        draw_p2 dp2(.clk(vclock),.hcount(hcount),.vcount(vcount),.blank(blank),
                .x(glove2pixelx),.y(glove2pixely),.ball_state(ball_state),
                .closed(glove2closed),.pixel(glove2pixel));







        reg[23:0] outpixel;
        reg hsync1delay;
        reg vsync1delay;
        reg blank1delay;
```

```
        reg hsync2delay;
        reg vsync2delay;
        reg blank2delay;
        reg hsync3delay;
        reg vsync3delay;
        reg blank3delay;
        reg hsync4delay;
        reg vsync4delay;
        reg blank4delay;
        reg hsync5delay;
        reg vsync5delay;
        reg blank5delay;
        reg hsync6delay;
        reg vsync6delay;
        reg blank6delay;

        always @(posedge vclock) begin
                hsync1delay <= hsync;
                vsync1delay <= vsync;
                blank1delay <= blank;
                hsync2delay <= hsync1delay;
                vsync2delay <= vsync1delay;
                blank2delay <= blank1delay;
                hsync3delay <= hsync2delay;
                vsync3delay <= vsync2delay;
                blank3delay <= blank2delay;
                hsync4delay <= hsync3delay;
                vsync4delay <= vsync3delay;
                blank4delay <= blank3delay;
                hsync5delay <= hsync4delay;
                vsync5delay <= vsync4delay;
                blank5delay <= blank4delay;
                hsync6delay <= hsync5delay;
                vsync6delay <= vsync5delay;
                blank6delay <= blank5delay;
                if (|ballpixel) outpixel <= ballpixel;
                else if (|glove1pixel) outpixel <= glove1pixel;
                else outpixel <= glove2pixel;
        end

    assign pixel = outpixel;

    assign phsync = hsync6delay;
    assign pvsync = vsync6delay;
    assign pblank = blank6delay;

endmodule
```

## global_coords.v

```
module global_coords(
    input clk,
    input rel_glove1x,
    input rel_glove1y,
    input rel_glove2x,
    input rel_glove2y,
    input[5:0] dist,//in meters
    input right_hand1,
    input right_hand2,
         input testright,
         input testleft,
         input testup,
         input testdown,
         input testright2,
         input testleft2,
         input testup2,
         input testdown2,
    output reg[15:0] glob_glove1x,
    output reg[15:0] glob_glove1y,
    output reg[15:0] glob_glove2x,
    output reg[15:0] glob_glove2y
    );

         //all distances except for dist are in millimeters; dist is in meters


         reg[17:0] update_counter; //210937

         ////////////////////////////////////
         //
         //button inputs until we get input from hand-tracking
         initial glob_glove1x = 16'd2000;
         initial glob_glove1y = 16'd2000;
         initial glob_glove2x = 16'd8000;
         initial glob_glove2y = 16'd2000;
         always @(posedge clk) begin

                if (update_counter == 0) begin
                        glob_glove1x <= glob_glove1x + (testright-testleft)*15;
                        glob_glove1y <= glob_glove1y + (testup-testdown)*15;
                        glob_glove2x <= glob_glove2x + (testright2-testleft2)*15;
                        glob_glove2y <= glob_glove2y + (testup2-testdown2)*15;
                        update_counter <= 210937;
                end else begin
                        glob_glove1x <= glob_glove1x;
                        glob_glove1y <= glob_glove1y;
```

```
                    glob_glove2x <= glob_glove2x;
                    glob_glove2y <= glob_glove2y;
                    update_counter <= update_counter-1;
            end
        end
        ////////////////////////////////////////


endmodule
```

## ballSM.v

```
module ballSM(
    input clk,
    input reset,
    input[15:0] glove1x,
    input[15:0] glove1y,
    input[15:0] glove2x,
    input[15:0] glove2y,
    input glove1closed,
    input glove2closed,
         input can_catch1,
         input can_catch2,
         input test0,
         input test1,
         input[5:0] dist,
         output debug,
         output reg catch_event,
         output reg throw_event,
    output reg[1:0] ball_state,//0 if ball is in the air,
                                                                    //1 if held by
glove1, 2 if held by glove2
    output reg[15:0] ball_x,
         output reg[15:0] ball_y
    );



         parameter updatesPerSec = 128;
         parameter tolerance = 300; //within how many mms a catch can be made
         parameter ballRadius = 50;

         //All distances in the inputs and outputs are in millimeters

         //Ball's velocity in millimeters/second
```

```
reg [15:0] ballvelx;
reg [15:0] ballvely;
reg ballvelxdir;//0 means right (positive x vel)
reg ballvelydir;//0 means up (positive y vel)


//This signal dictates when velocity and position are updated
reg update;
reg[18:0] update_counter;

//These variables keep track of past positions of the ball to determine
//         its velocity
reg[15:0] pastposx;
reg[15:0] pastposy;

//These keep track of whether the glove has recently been opened
reg glove1opened;
reg glove2opened;
reg glove1edge;
reg glove2edge;

//These keep track of whether the ball is close enough to a glove to get caught
wire closeToGlove1;
wire closeToGlove2;


wire[15:0] mmdist;


//This keeps track of whether the ball is touching the floor or a wall
wire ballAtEdge;

       assign debug = ~closeToGlove1;
       assign mmdist = {10'b0,dist}*1000;
        assign closeToGlove1 =
              ((ball_x >= glove1x && ball_x - glove1x < tolerance)
              || (ball_x < glove1x && glove1x - ball_x < tolerance)) &&
              ((ball_y >= glove1y && ball_y - glove1y < tolerance)
              || (ball_y < glove1y && glove1y - ball_y < tolerance));
        assign closeToGlove2 =
              ((ball_x >= glove2x && ball_x - glove2x < tolerance)
              || (ball_x < glove2x && glove2x - ball_x < tolerance)) &&
              ((ball_y >= glove2y && ball_y - glove2y < tolerance)
              || (ball_y < glove2y && glove2y - ball_y < tolerance));
        assign ballAtEdge =
              (ball_x < ballRadius + 5) || (ball_x > mmdist + 4000 - ballRadius)
              || (ball_y < ballRadius + 5);
```

```
	reg[3:0] trig_counter;

always @(posedge clk) begin
	if (catch_event) trig_counter <= 5;
	if (throw_event) trig_counter <= 5;
	if (trig_counter == 0) begin
		catch_event <= 0;
		throw_event <= 0;
	end else trig_counter <= trig_counter - 1;
	glove1opened <= ~glove1closed;
	glove2opened <= ~glove2closed;
	glove1edge <= glove1closed && glove1opened;
	glove2edge <= glove2closed && glove2opened;
	if (update_counter == 0) begin
		update <= 1;
		update_counter <= 507811;//128Hz must be same as updatesPerSec
	end else begin
		update <= 0;
		update_counter <= update_counter - 1;
	end
	//if reset is asserted and the ball is in the air, it appears
	//in one of the gloves if either of them is closed
	if (reset) begin
		if (ball_state == 0) begin
			if (glove1closed) begin
				ball_state <= 1;
				ball_x <= glove1x;
				pastposx <= glove1x;
				ball_y <= glove1y;
				pastposy <= glove1y;
				ballvelx <= 0;
				ballvely <= 0;
			end else if (glove2closed) begin
				ball_state <= 2;
				ball_x <= glove2x;
				pastposx <= glove1x;
				ball_y <= glove2y;
				pastposy <= glove1y;
				ballvelx <= 0;
				ballvely <= 0;
			end
		end
	end else if (test0) begin
		ball_state <= 0;
		ball_x <= glove1x;
		ball_y <= glove1y;
		ballvelx <= 0;
```

```
                ballvely <= 4000;
                ballvelydir <= 0;
        end else if (test1) begin
                ball_state <= 0;
                ball_x <= glove1x;
                ball_y <= glove1y;
                ballvelx <= 4000;
                ballvely <= 4000;
                ballvelxdir <= 0;
                ballvelydir <= 0;
        end else begin
                if (update) begin
                        pastposx <= ball_x;
                        pastposy <= ball_y;
                        if (ball_state > 0) begin
                                if (ball_x > pastposx)begin
                                        ballvelx <= (ball_x - pastposx)*updatesPerSec;
                                        ballvelxdir <= 0;
                                end else begin
                                        ballvelx <= (pastposx - ball_x)*updatesPerSec;
                                        ballvelxdir <= 1;
                                end
                                if (ball_y > pastposy)begin
                                        ballvely <= (ball_y - pastposy)*updatesPerSec;
                                        ballvelydir <= 0;
                                end else begin
                                        ballvely <= (pastposy - ball_y)*updatesPerSec;
                                        ballvelydir <= 1;
                                end
                        end else begin
                                ballvelx <= ballvelx; //no air resistance
                                if (ballvelydir == 0) begin
                                //77 = g*DeltaT = 9806 mm/s^2 * (1/128 s)
                                        if (ballvely >= 77) ballvely <= ballvely - 77;
                                        else begin
                                                ballvelydir <= 1;
                                                ballvely <= 77 - ballvely;
                                        end
                                end else begin
                                        if (ballvely <= 16'hFFFF - 16'd77) ballvely <=
ballvely + 77;
                                        else ballvely <= 16'hFFFF;
                                end
                        end
                end
                case (ball_state)
                        0:
                        begin
```

```
                                        if (update) begin
                                                if (ballAtEdge) begin
                                                        ball_x <= ball_x;
                                                        ball_y <= ball_y;
                                                end else begin
                                                //DeltaX = v*DeltaT
                                                        if (~ballvelxdir) ball_x <= ball_x +
(ballvelx / updatesPerSec);
                                                        else ball_x <= ball_x - (ballvelx /
updatesPerSec);
                                                        if (~ballvelydir) ball_y <= ball_y +
(ballvely / updatesPerSec);
                                                        else begin
                                                                if (ball_y > (ballvely /
updatesPerSec)) ball_y <= ball_y - (ballvely / updatesPerSec);
                                                                else ball_y <= 5;
                                                        end
                                                end
                                        end
                                        if (glove1edge && can_catch1 && closeToGlove1) begin
                                                ball_state <= 1;
                                                catch_event <= 1;
                                        end else if (glove2edge && can_catch2 &&
closeToGlove2) begin
                                                ball_state <= 2;
                                                catch_event <= 1;
                                        end else ball_state <= 0;
                                end
                                1:
                                begin
                                        ball_x <= glove1x;
                                        ball_y <= glove1y;
                                        if (glove1closed) ball_state <= 1;
                                        else begin
                                                ball_state <= 0;
                                                throw_event <= 1;
                                        end
                                end
                                2:
                                begin
                                        ball_x <= glove2x;
                                        ball_y <= glove2y;
                                        if (glove2closed) ball_state <= 2;
                                        else begin
                                                ball_state <= 0;
                                                throw_event <= 1;
                                        end
                                end
```

```
                        default:
                        begin
                                ball_x <= ball_x;
                                ball_y <= ball_y;
                        end
                endcase
        end
    end

    //for testing purposes
    initial ball_state = 0;
    initial ball_x = 16'd4000;
    initial ball_y = 16'd2000;
    initial ballvelx = 500;
    initial ballvely = 3000;
    initial ballvelxdir = 0;
    initial ballvelydir = 0;


endmodule
```

## coords_to_pixel.v

```
module coords_to_pixel(
    input[15:0] x_coord,//in mm
    input[15:0] y_coord,//in mm
    input[5:0] dist,//in meters
    output reg[10:0] pixel_x,
    output reg[9:0] pixel_y
    );

    wire[6:0] maxdist;
    //negative coords or coords greater than maxdist will not be on screen
    assign maxdist = dist + 3;

    reg[2:0] powerOf2;



    always @(*) begin
        if (maxdist < 4) powerOf2 = 2;
        else if (maxdist < 8) powerOf2 = 3;
        else if (maxdist < 16) powerOf2 = 4;
        else if (maxdist < 32) powerOf2 = 5;
        else powerOf2 = 6;
        //multiply by 1024 (number of pixels) and divide by 1024 for mm's per 1.024m
        //so do nothing
```

```
            pixel_x = x_coord >> powerOf2;
            pixel_y = 767 - (y_coord >> powerOf2);
        end


endmodule
```

## draw_ball.v

```
module draw_ball(
        input clk,
    input[10:0] hcount,
    input[9:0] vcount,
    input blank,
         input[15:0] x,
         input[15:0] y,
         input[1:0] ball_state,
    output reg[23:0] pixel
    );

         wire[11:0] map_addr;//memory address in color map
         reg[11:0] map_addr2;
         wire[3:0] table_addr;//memory address in color table
         reg[3:0] table_addr2;
         wire[23:0] prepixel;

         //get map address
         get_map_address gma(.clk(clk),.hcount(hcount),.vcount(vcount),.blank(blank),
                                                .x(x),.y(y),.addr(map_addr));


         //get table address
         ball_color_map map(.clka(clk),.addra(map_addr2),.douta(table_addr));

         //get table entry
         ball_color_table ctable(.clka(clk),.addra(table_addr2),.douta(prepixel));

         always @(posedge clk) begin
                map_addr2<=map_addr;
                table_addr2<=table_addr;
                if (ball_state == 0) pixel <= prepixel;
                else pixel <= 24'd0;
         end
```

```
endmodule
```

## draw_p1.v

```
module draw_p1(
        input clk,
    input[10:0] hcount,
    input[9:0] vcount,
    input blank,
         input[15:0] x,
         input[15:0] y,
         input[1:0] ball_state,
         input closed,
    output reg[23:0] pixel
    );

         wire[11:0] map_addr;//memory address in color map
         reg[11:0] map_addr2;
         wire[3:0] table_addr;//memory address in color table
         wire[3:0] table_addrh;
         wire[3:0] table_addrc;
         reg[3:0] table_addr2;
         reg[3:0] table_addrh2;
         reg[3:0] table_addrc2;
         wire[23:0] prepixel1;
         wire[23:0] prepixel2;
         wire[23:0] prepixel3;

         //get map address
         get_map_address gma(.clk(clk),.hcount(hcount),.vcount(vcount),.blank(blank),
                                                .x(x),.y(y),.addr(map_addr));


         //get table address
         open_hand_l_color_map map(.clka(clk),.addra(map_addr2),.douta(table_addr));
         holding_ball_l_color_map hmap(.clka(clk),.addra(map_addr2),.douta(table_addrh));
         closed_hand_l_color_map cmap(.clka(clk),.addra(map_addr2),.douta(table_addrc));

         //get table entry
         open_hand_color_table ctable(.clka(clk),.addra(table_addr2),.douta(prepixel1));
         holding_ball_l_color_table
hctable(.clka(clk),.addra(table_addrh2),.douta(prepixel2));
         closed_hand_color_table cctable(.clka(clk),.addra(table_addrc2),.douta(prepixel3));

         always @(posedge clk) begin
```

```
            map_addr2<=map_addr;
            table_addr2<=table_addr;
            table_addrh2<=table_addrh;
            table_addrc2<=table_addrc;
            if (ball_state == 1) pixel <= prepixel2;
            else if (closed) pixel <= prepixel3;
            else pixel <= prepixel1;
        end



endmodule
```

## draw_p2.v

```
module draw_p2(
        input clk,
    input[10:0] hcount,
    input[9:0] vcount,
    input blank,
         input[15:0] x,
         input[15:0] y,
         input[1:0] ball_state,
         input closed,
    output reg[23:0] pixel
    );

         wire[11:0] map_addr;//memory address in color map
         reg[11:0] map_addr2;
         wire[3:0] table_addr;//memory address in color table
         wire[3:0] table_addrh;
         wire[3:0] table_addrc;
         reg[3:0] table_addr2;
         reg[3:0] table_addrh2;
         reg[3:0] table_addrc2;
         wire[23:0] prepixel1;
         wire[23:0] prepixel2;
         wire[23:0] prepixel3;

         //get map address
         get_map_address gma(.clk(clk),.hcount(hcount),.vcount(vcount),.blank(blank),
                                                 .x(x),.y(y),.addr(map_addr));
```

```
        //get table address
        open_hand_color_map map(.clka(clk),.addra(map_addr2),.douta(table_addr));
        holding_ball_color_map hmap(.clka(clk),.addra(map_addr2),.douta(table_addrh));
        closed_hand_color_map cmap(.clka(clk),.addra(map_addr2),.douta(table_addrc));

        //get table entry
        open_hand_color_table ctable(.clka(clk),.addra(table_addr2),.douta(prepixel1));
        holding_ball_color_table hctable(.clka(clk),.addra(table_addrh2),.douta(prepixel2));
        closed_hand_color_table cctable(.clka(clk),.addra(table_addrc2),.douta(prepixel3));

        always @(posedge clk) begin
               map_addr2<=map_addr;
               table_addr2<=table_addr;
               table_addrh2<=table_addrh;
               table_addrc2<=table_addrc;
               if (ball_state == 2) pixel <= prepixel2;
               else if (closed) pixel <= prepixel3;
               else pixel <= prepixel1;
        end



endmodule
```

## get_map_address.v

```
module get_map_address(
    input clk,
    input[10:0] hcount,
    input[9:0] vcount,
    input blank,
    input[15:0] x,
    input[15:0] y,
    output reg[11:0] addr
    );


        wire outofbounds1;
        wire outofbounds2;
        wire outofbounds3;
        wire outofbounds4;
        reg outofbounds;

        reg[15:0] fulladdr;
```

```
        reg[15:0] dx;
        reg[15:0] dy;


        parameter xoffset = 16'd35;
        parameter yoffset = 16'd25;



                //determines whether current pixel is within object sprite
                assign outofbounds1 = ((hcount < x) && hcount < x - xoffset+1);
                assign outofbounds2 = (hcount > x && hcount > x + xoffset-1);
                assign outofbounds3 = (vcount < y && vcount < y - yoffset+1);
                assign outofbounds4 = (vcount > y && vcount > y + yoffset-1);


        always @(posedge clk) begin
                outofbounds <= blank || outofbounds1 || outofbounds2 || outofbounds3 ||
outofbounds4;
                dx <= (hcount + xoffset - x);
                dy <= (vcount + yoffset - y)*70;
                if (outofbounds) begin
                        fulladdr <= 0;
                end else begin
                        fulladdr <=  dy + dx;
                end
                addr <= fulladdr[11:0];
        end




endmodule
```

## sound.v

```
module sound(
                input clk,
                input trigger,
                input ready,
                output reg[7:0] data);

                reg[13:0] address = 0;
```

```
                reg playing;
                wire [7:0] predata;


                soundrom sr(.clka(clk),.addra(address),.douta(predata));

                always @(posedge clk) begin
                        if (trigger && ~playing) begin
                                playing <= 1;
                                address <= 0;


                                data <= 0;
                        end else if (playing) begin
                           if (address == 8999) playing <= 0;
                                else if (ready) begin
                                        address <= address + 1;

                                end

                                data <= predata;
                        end
                        else data <= 0;
                end

endmodule




//
// bi-directional monaural interface to AC97
//
////////////////////////////////////////////////////////////////////////

module lab5audio (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  //output wire [7:0] audio_in_data,
  input wire [7:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);
```

```
      wire [7:0] audio_in_data;

wire [7:0] command_address;
wire [15:0] command_data;
wire command_valid;
wire [19:0] left_in_data, right_in_data;
wire [19:0] left_out_data, right_out_data;

// wait a little before enabling the AC97 codec
reg [9:0] reset_count;
always @(posedge clock_27mhz) begin
  if (reset) begin
    audio_reset_b = 1'b0;
    reset_count = 0;
  end else if (reset_count == 1023)
    audio_reset_b = 1'b1;
  else
    reset_count = reset_count+1;
end

wire ac97_ready;
ac97 ac97(.ready(ac97_ready),
          .command_address(command_address),
          .command_data(command_data),
          .command_valid(command_valid),
          .left_data(left_out_data), .left_valid(1'b1),
          .right_data(right_out_data), .right_valid(1'b1),
          .left_in_data(left_in_data), .right_in_data(right_in_data),
          .ac97_sdata_out(ac97_sdata_out),
          .ac97_sdata_in(ac97_sdata_in),
          .ac97_synch(ac97_synch),
          .ac97_bit_clock(ac97_bit_clock));

// ready: one cycle pulse synchronous with clock_27mhz
reg [2:0] ready_sync;
always @ (posedge clock_27mhz) ready_sync <= {ready_sync[1:0], ac97_ready};
assign ready = ready_sync[1] & ~ready_sync[2];

reg [7:0] out_data;
always @ (posedge clock_27mhz)
  if (ready) out_data <= audio_out_data;
assign audio_in_data = left_in_data[19:12];
assign left_out_data = {out_data, 12'b000000000000};
assign right_out_data = left_out_data;

// generate repeating sequence of read/writes to AC97 registers
ac97commands cmds(.clock(clock_27mhz), .ready(ready),
```

```
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (
  output reg ready,
  input wire [7:0] command_address,
  input wire [15:0] command_data,
  input wire command_valid,
  input wire [19:0] left_data,
  input wire left_valid,
  input wire [19:0] right_data,
  input wire right_valid,
  output reg [19:0] left_in_data, right_in_data,
  output reg ac97_sdata_out,
  input wire ac97_sdata_in,
  output reg ac97_synch,
  input wire ac97_bit_clock
);
  reg [7:0] bit_count;

  reg [19:0] l_cmd_addr;
  reg [19:0] l_cmd_data;
  reg [19:0] l_left_data, l_right_data;
  reg l_cmd_v, l_left_v, l_right_v;

  initial begin
    ready <= 1'b0;
    // synthesis attribute init of ready is "0";
    ac97_sdata_out <= 1'b0;
    // synthesis attribute init of ac97_sdata_out is "0";
    ac97_synch <= 1'b0;
    // synthesis attribute init of ac97_synch is "0";

    bit_count <= 8'h00;
    // synthesis attribute init of bit_count is "0000";
    l_cmd_v <= 1'b0;
    // synthesis attribute init of l_cmd_v is "0";
    l_left_v <= 1'b0;
    // synthesis attribute init of l_left_v is "0";
    l_right_v <= 1'b0;
    // synthesis attribute init of l_right_v is "0";

    left_in_data <= 20'h00000;
```

```
  // synthesis attribute init of left_in_data is "00000";
  right_in_data <= 20'h00000;
  // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
  // Generate the sync signal
  if (bit_count == 255)
    ac97_synch <= 1'b1;
  if (bit_count == 15)
    ac97_synch <= 1'b0;

  // Generate the ready signal
  if (bit_count == 128)
    ready <= 1'b1;
  if (bit_count == 2)
    ready <= 1'b0;

  // Latch user data at the end of each frame. This ensures that the
  // first frame after reset will be empty.
  if (bit_count == 255) begin
    l_cmd_addr <= {command_address, 12'h000};
    l_cmd_data <= {command_data, 4'h0};
    l_cmd_v <= command_valid;
    l_left_data <= left_data;
    l_left_v <= left_valid;
    l_right_data <= right_data;
    l_right_v <= right_valid;
  end

  if ((bit_count >= 0) && (bit_count <= 15))
    // Slot 0: Tags
    case (bit_count[3:0])
      4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
      4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
      4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
      4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
      4'h4: ac97_sdata_out <= l_right_v; // Right data valid
      default: ac97_sdata_out <= 1'b0;
    endcase
  else if ((bit_count >= 16) && (bit_count <= 35))
    // Slot 1: Command address (8-bits, left justified)
    ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
  else if ((bit_count >= 36) && (bit_count <= 55))
    // Slot 2: Command data (16-bits, left justified)
    ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
  else if ((bit_count >= 56) && (bit_count <= 75)) begin
    // Slot 3: Left channel
```

```
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
    end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;
  end // always @ (posedge ac97_bit_clock)

  always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
  end
endmodule

// issue initialization commands to AC97
module ac97commands (
  input wire clock,
  input wire ready,
  output wire [7:0] command_address,
  output wire [15:0] command_data,
  output reg command_valid,
  input wire [4:0] volume,
  input wire [2:0] source
);
  reg [23:0] command;

  reg [3:0] state;
  initial begin
    command <= 4'h0;
    // synthesis attribute init of command is "0";
    command_valid <= 1'b0;
    // synthesis attribute init of command_valid is "0";
    state <= 16'h0000;
    // synthesis attribute init of state is "0000";
  end

  assign command_address = command[23:16];
  assign command_data = command[15:0];

  wire [4:0] vol;
```

```
  assign vol = 31-volume;  // convert to attenuation

  always @(posedge clock) begin
    if (ready) state <= state+1;

    case (state)
      4'h0: // Read ID
        begin
          command <= 24'h80_0000;
          command_valid <= 1'b1;
        end
      4'h1: // Read ID
        command <= 24'h80_0000;
      4'h3: // headphone volume
        command <= { 8'h04, 3'b000, vol, 3'b000, vol };
      4'h5: // PCM volume
        command <= 24'h18_0808;
      4'h6: // Record source select
        command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
      4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
      4'h9: // set +20db mic gain
        command <= 24'h0E_8048;
      4'hA: // Set beep volume
        command <= 24'h0A_0000;
      4'hB: // PCM out bypass mix1
        command <= 24'h20_8000;
      default:
        command <= 24'h80_0000;
    endcase // case(state)
  end // always @ (posedge clock)
endmodule // ac97commands
```